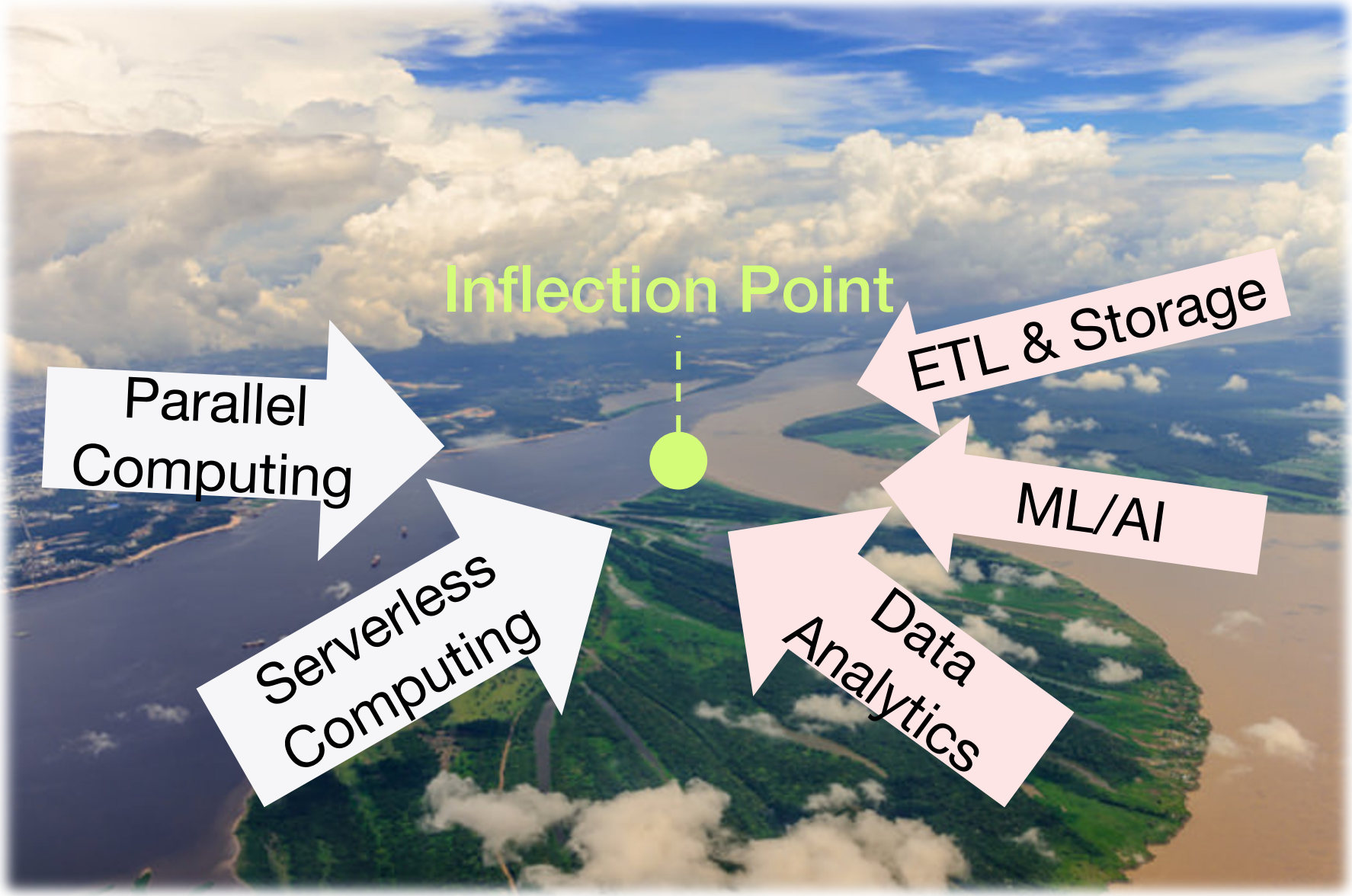
Serverless Parallel Computing

DS 5110: Big Data Systems (Spring 2023)

Lecture 7b

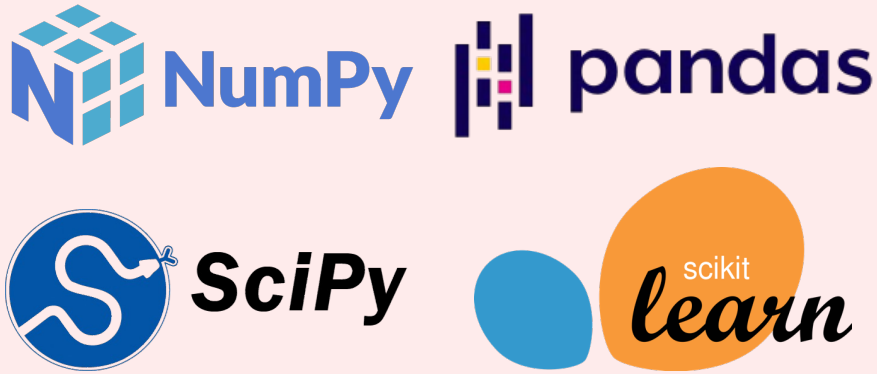
Yue Cheng

Confluence: When stateful apps meet serverless computing



Today's data analytics landscape

Libraries efficient for O(1MB)

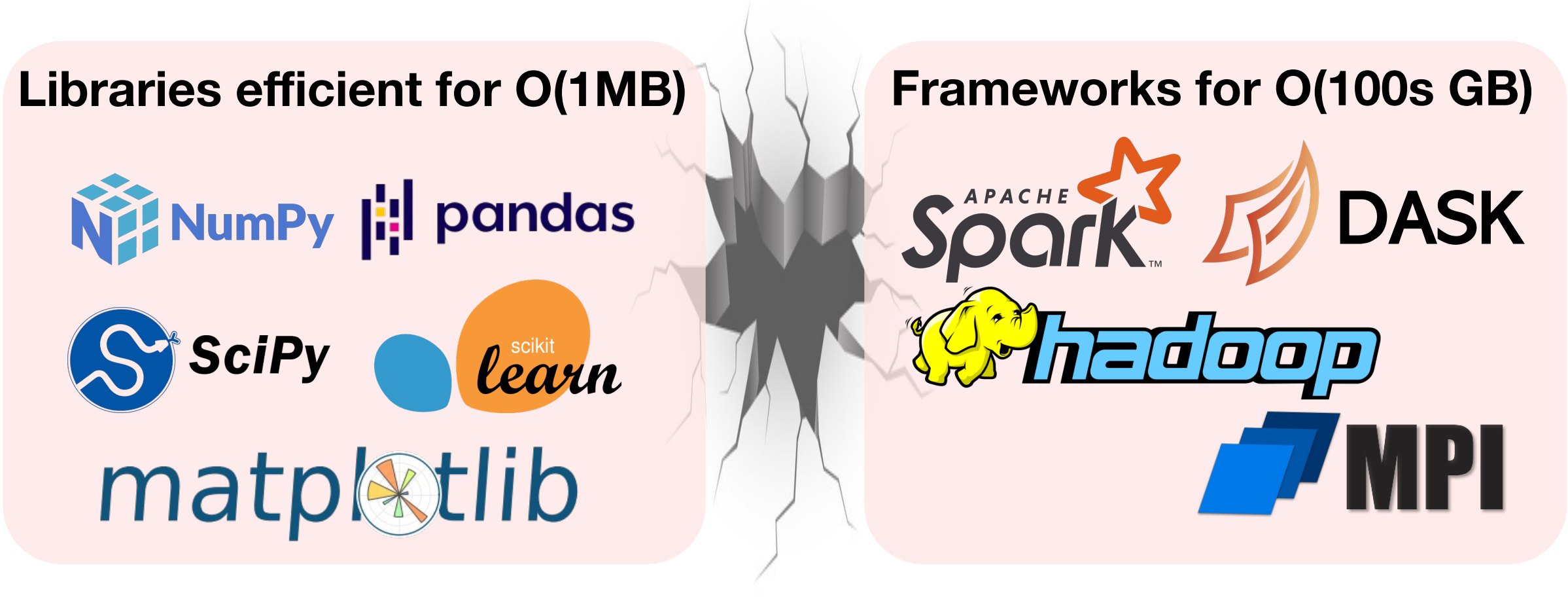


Today's data analytics landscape

Libraries efficient for O(1MB)

Frameworks for O(100s GB)

Today's data analytics landscape



Today's data analytics landscape

Libraries efficient for O(1MB)

- Easy to program (writing centralized code)
- Low barrier for environment setup (just installing libs)
- Well understood

- No scalability / elasticity
- Not able to efficiently handle large data

Frameworks for O(100s GB)

- Scale to 100s GB data

- Difficult to program and debug
  - Requires distributed systems knowledge
- No elasticity
- High barrier for environment setup
  - Requires low-level administration skills

Today's data analytics landscape

Libraries efficient for O(1MB)

Easy-to-use but not scalable nor elastic

Frameworks for O(100s GB)

Scalable but not easy-to-use nor elastic

Making a strong case for

Running elastic, pay-per-use stateful apps on Serverless

Libraries efficient for O(1MB)

Frameworks for O(100s GB)

Easy-to-use

Elastic

Pay-per-use

Scalable

Recap: Serverless computing

Recap: Serverless computing

Car analogy

Own a car

(Bare metal servers)

Rent a car

(VPS)

City car-sharing

(Serverless)

Cars are parked **95%** of the time (loige.link/car-parked-95)

How much do you use the car?

Recap: What is serverless computing?

Many people define it many ways

A programming abstraction that enables users to upload programs, run them at **virtually** any scale, and pay **only for the resources used**

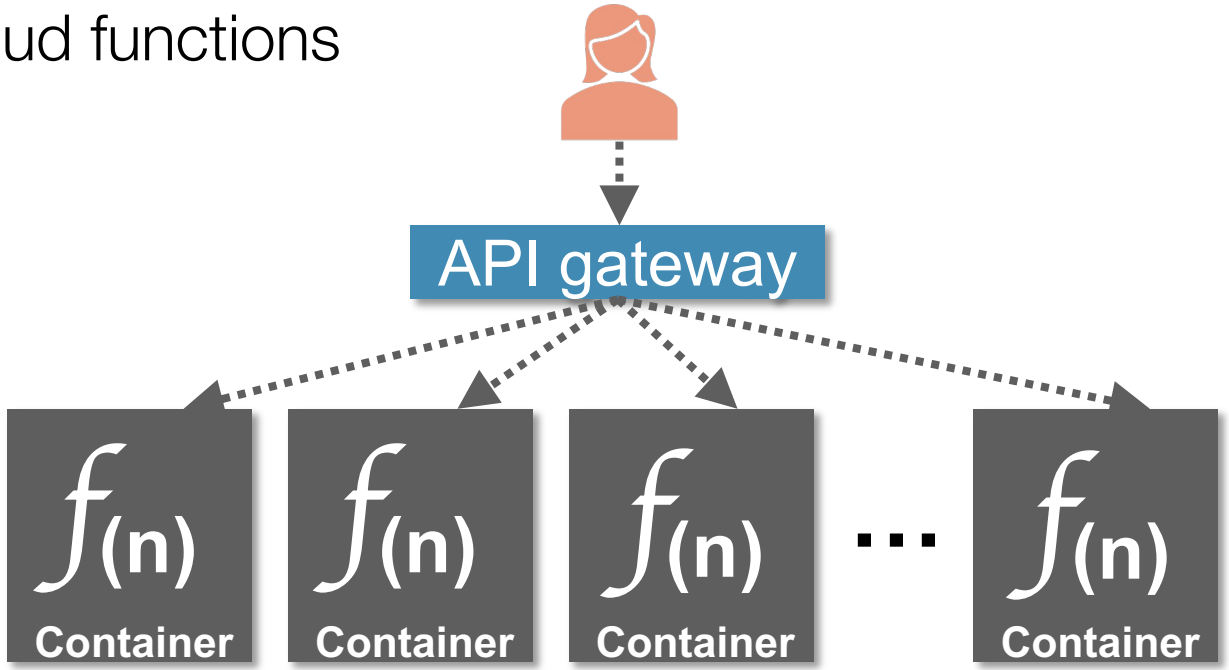
- **Function-as-a-Service (FaaS):** Cloud functions as a basic deployment unit

AWS Lambda

Azure Functions

Google Cloud Functions

Alibaba Function Compute

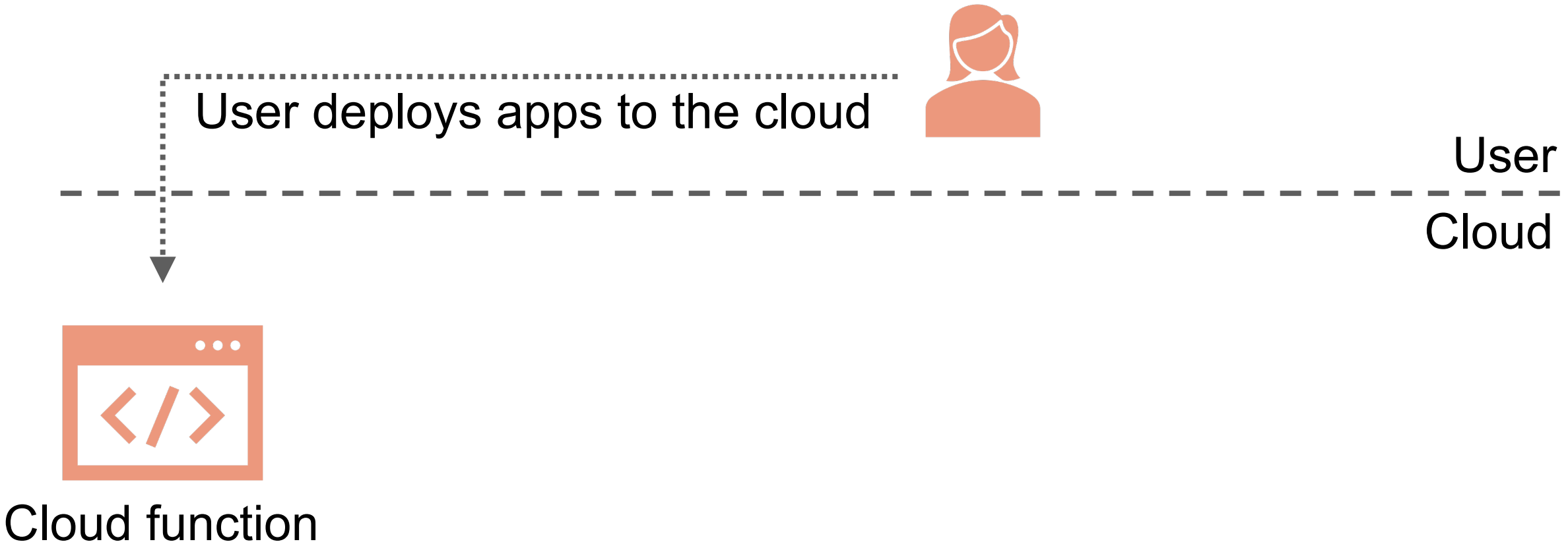


Function-as-a-Service (FaaS)

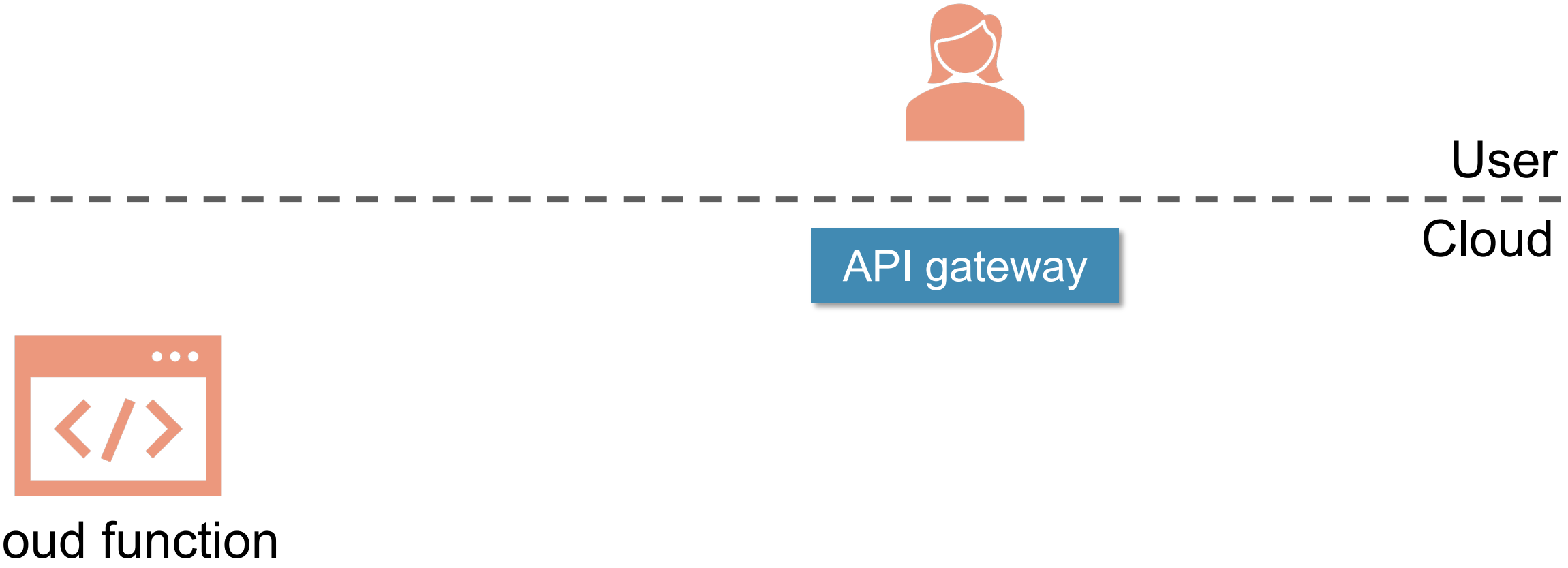
User

Cloud

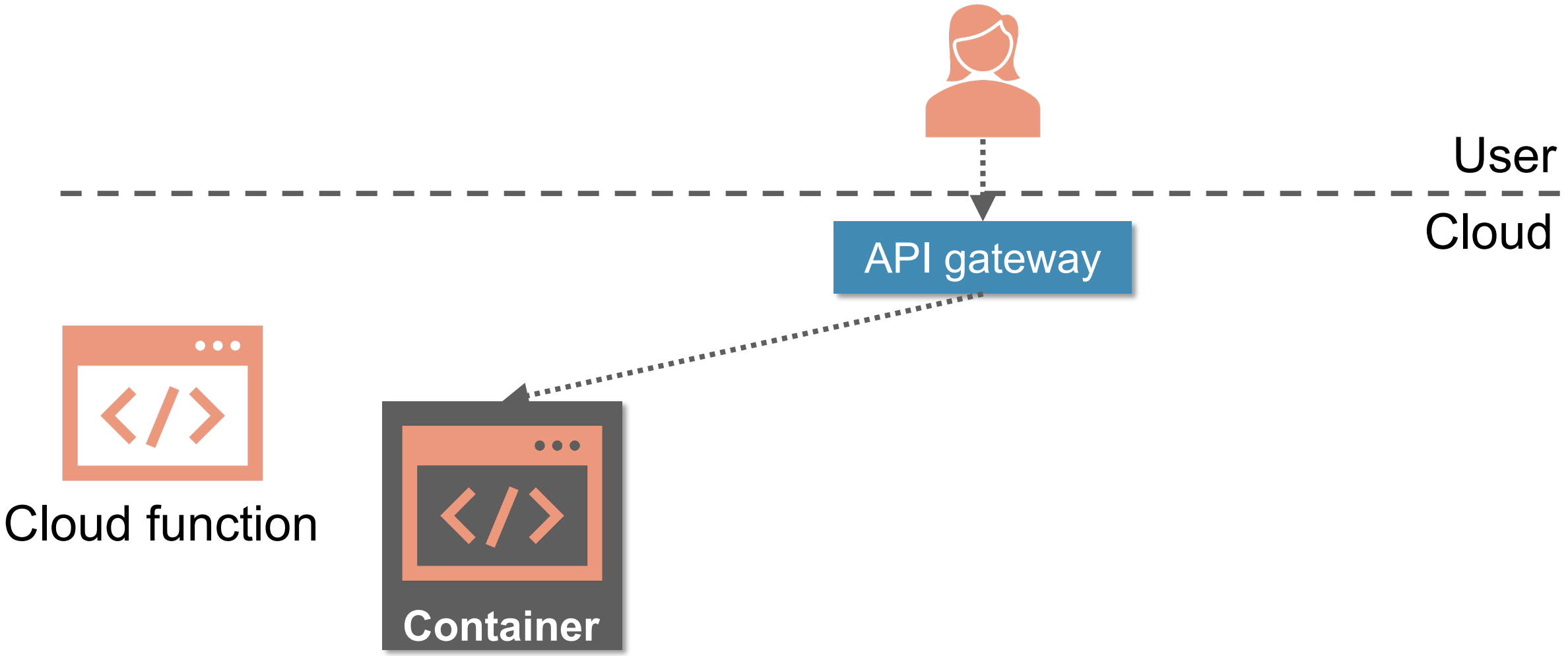
Function-as-a-Service (FaaS)



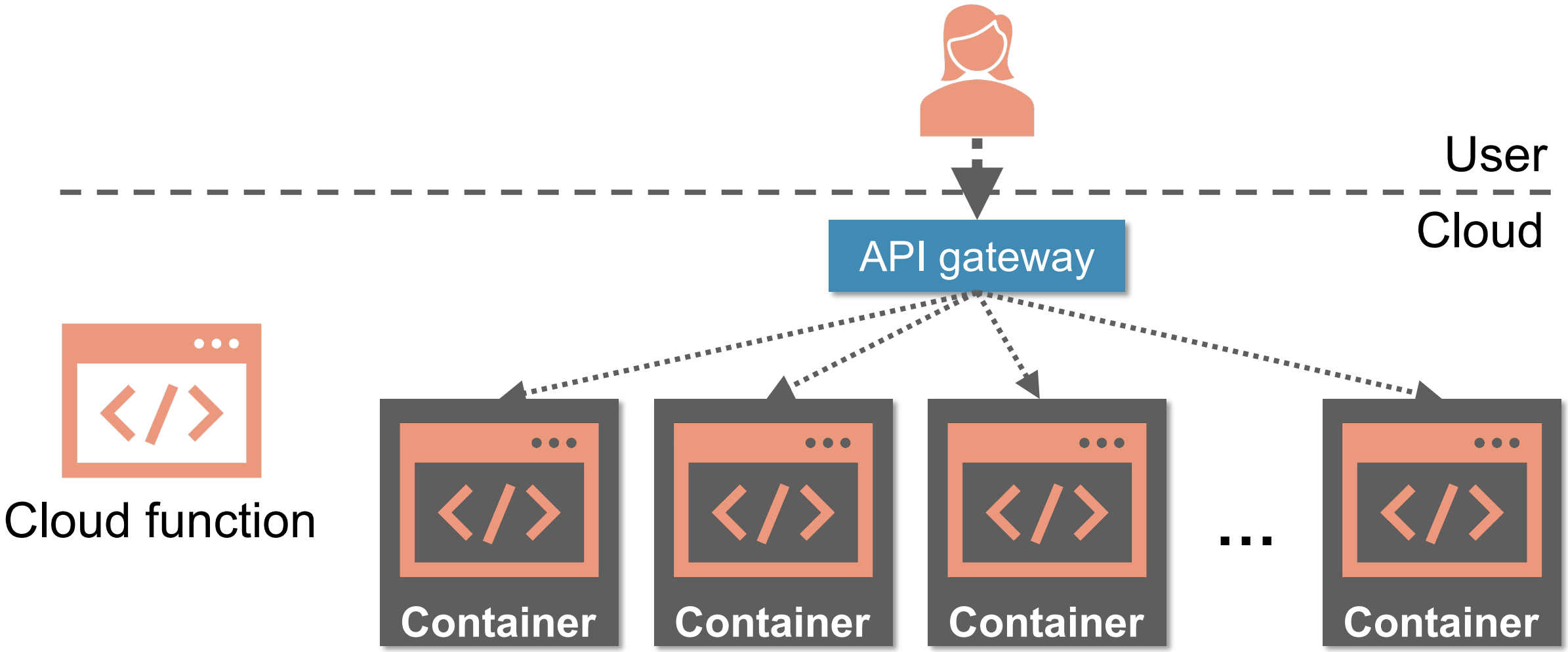
Function-as-a-Service (FaaS)



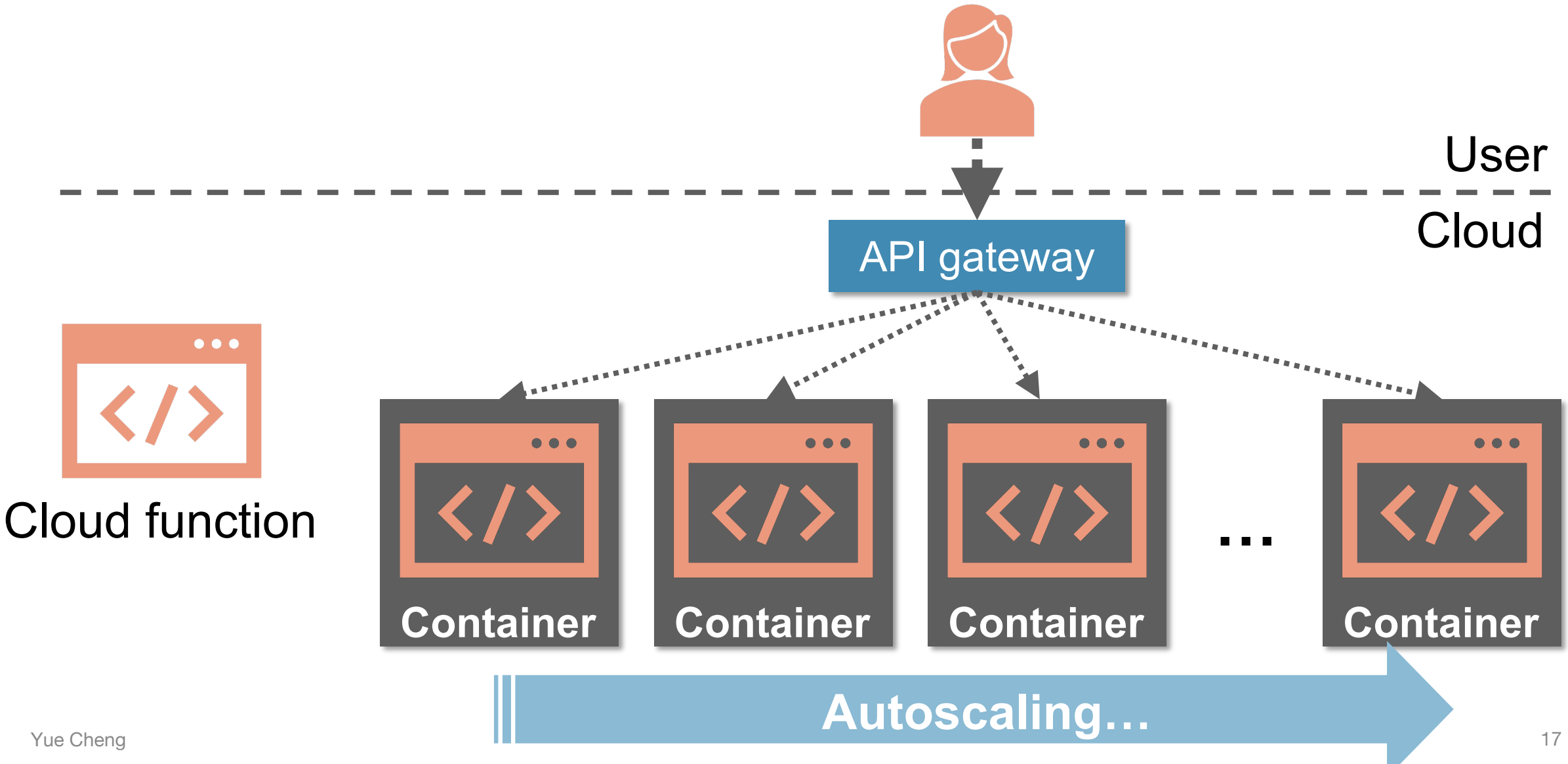
Function-as-a-Service (FaaS)



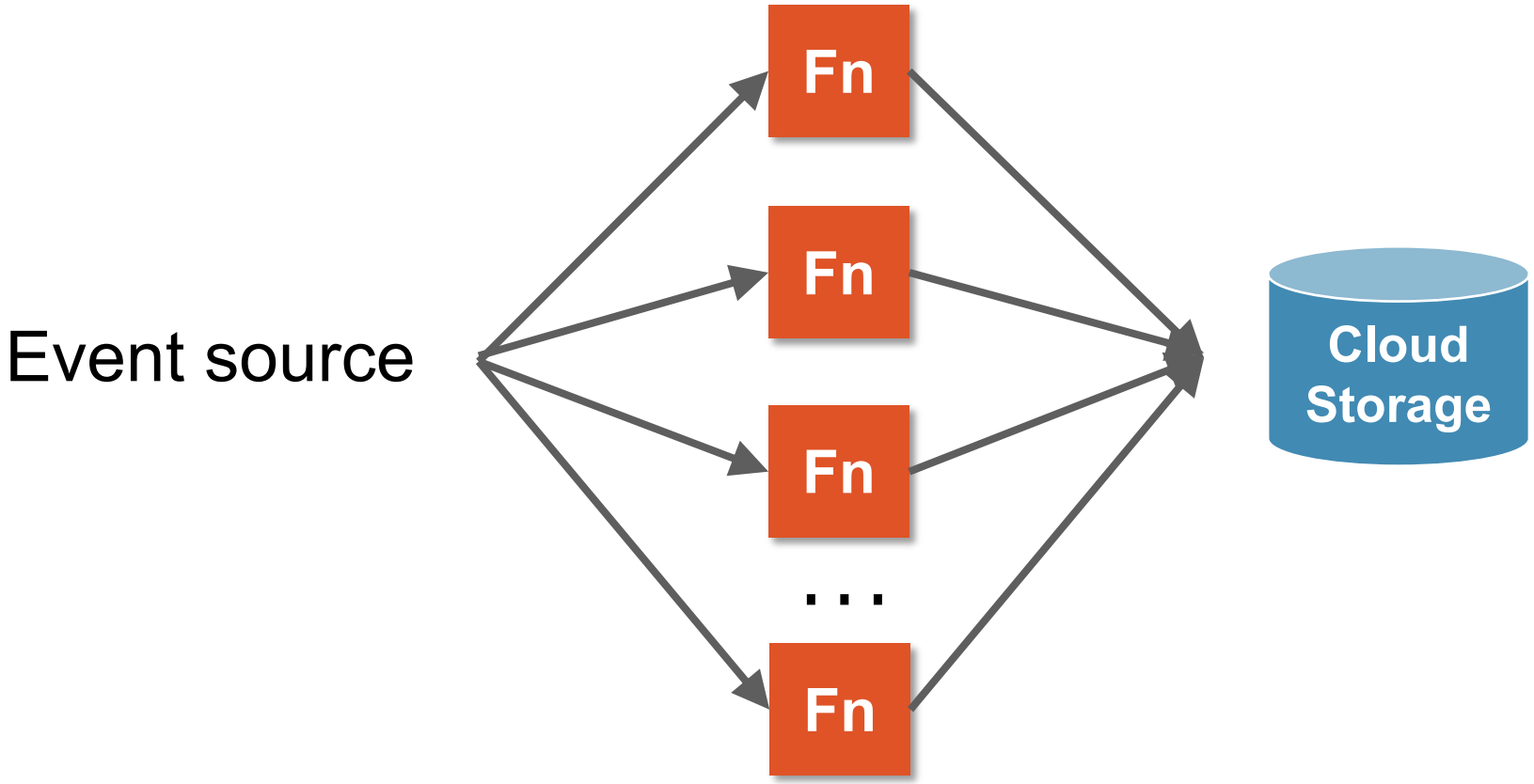
Function-as-a-Service (FaaS)



Function-as-a-Service (FaaS)



What is FaaS good at today?



Embarrassingly parallel tasks
Stateless processing
...

Limitations of FaaS today

- No guaranteed data availability
- Banned inbound network
- Limited per-function resources
- Limited function execution time

Limitations of FaaS today

- No guaranteed data availability

- Banned inbound network
- Limited per-function resources
- Limited function execution time

⚠ Cloud functions could be reclaimed any time

⚠ In-memory state is lost

Limitations of FaaS today

- No guaranteed data availability
- Banned inbound network
- Limited per-function resources
- Limited function execution time

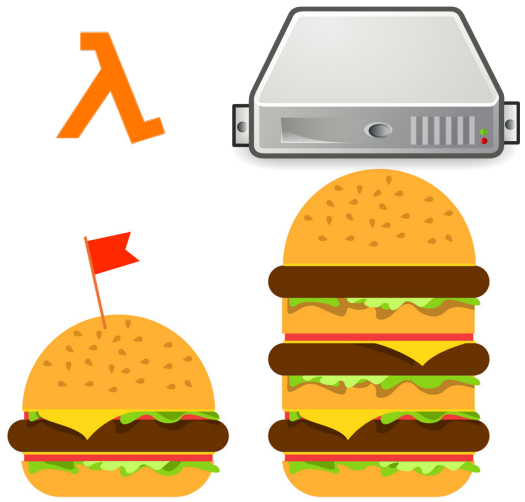
⚠ Cloud functions cannot run as a server

Limitations of FaaS today

- No guaranteed data availability
- Banned inbound network
- Limited per-function resources
- Limited function execution time

⚠ Limited CPU & memory

⚠ I/O is a bottleneck



Limitations of FaaS today

- No guaranteed data availability
- Banned inbound network
- Limited per-function resources
- Limited function execution time

⚠ Limited to up to 15 min

Challenges of supporting stateful apps on FaaS

Research Question: Is FaaS poorly suited for stateful applications because these applications share state?

Case studies:

1. **[Programming model]** How to design FaaS-centric parallel computing to **enable easy programming of 10,000 CPU cores and 15,000 GBs of RAM**?
2. **[Data storage]** How to exploit FaaS **elasticity and pay-per-use** to reduce the $$ cost by **100X**?

Challenges of supporting stateful apps on FaaS

Research Question: Is FaaS poorly suited for stateful applications because these applications share state?

Case studies:

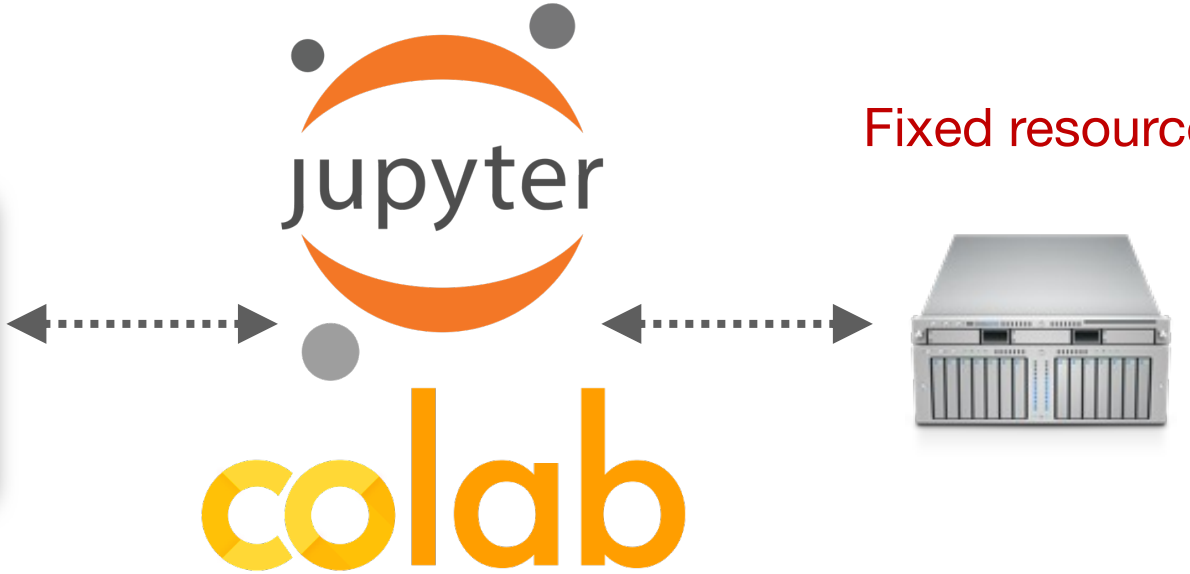
1. **[Programming model]** How to design FaaS-centric parallel computing to **enable easy programming of 10,000 CPU cores and 15,000 GBs of RAM**? **← Today**
2. **[Data storage]** How to exploit FaaS **elasticity and pay-per-use** to reduce the $$ cost by **100X**?

Python analytics: What we have today

```
def MyFunc(X, Y):
  return np.matmul(X, Y)
```

Python analytics: What we have today

```
def MyFunc(X, Y):
  return np.matmul(X, Y)
```



Fixed resource

User writes interactive analytics and runs it on a notebook server

- No autoscaling for large computations
- Too slow? OOM? Need to scale out manually!
- Too expensive? Idled resources charge $$

Python analytics: What we have today

```
def MyFunc(X, Y):
  return np.matmul(X, Y)
```

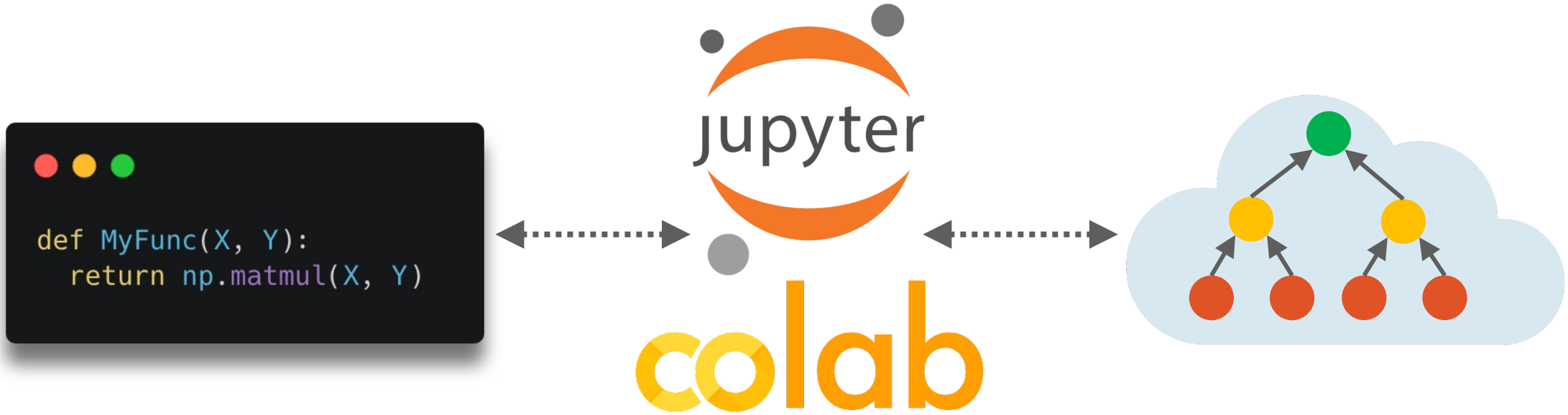
Manually deployed
distributed computing cluster

User writes interactive analytics and runs it on a notebook server

- No autoscaling for large computations
- Too slow? OOM? Need to scale out manually!
- Too expensive? Idled resources charge $$

High barriers to enter for those who lack CS/systems background

Python analytics: What we would like to have



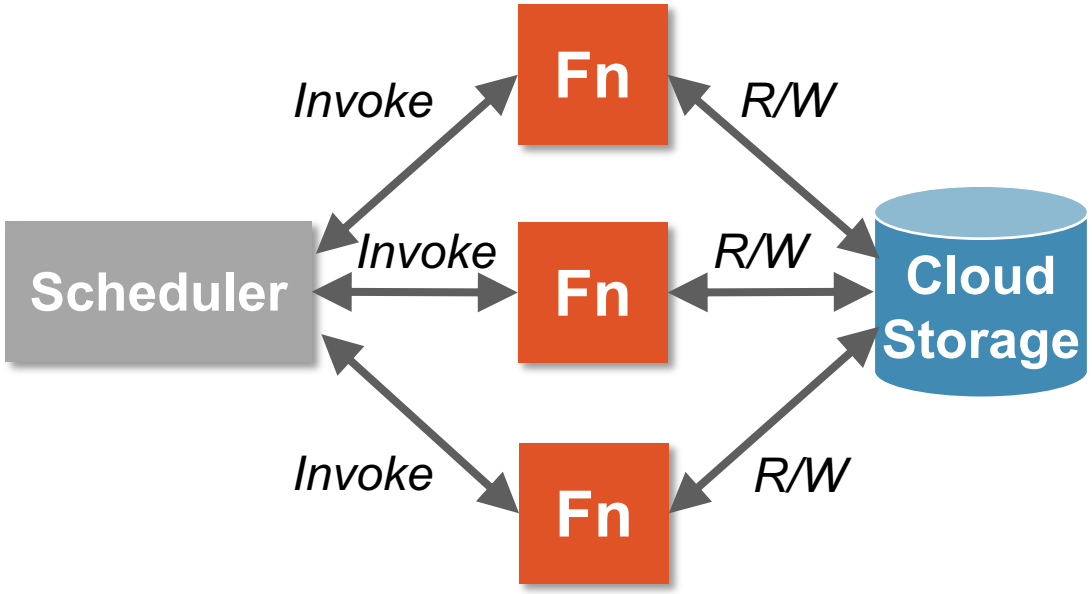
User writes interactive analytics and runs it **on FaaS**

- Elastically & automatically scales to right size
- Pay-per-use with minimal $$ cost
- Expertise of writing parallel programs **NOT required**
- Manual cluster maintenance **NOT required**

Quantifying the pain of FaaS

Or, how FaaS adds huge amounts of **performance taxes**

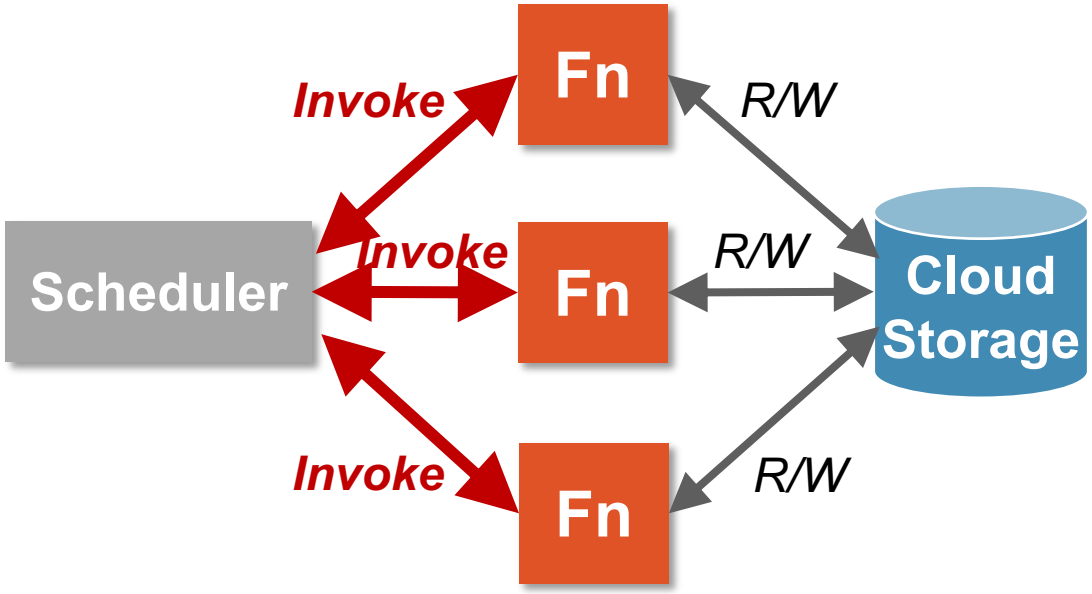
Python analytics on FaaS is too slow!

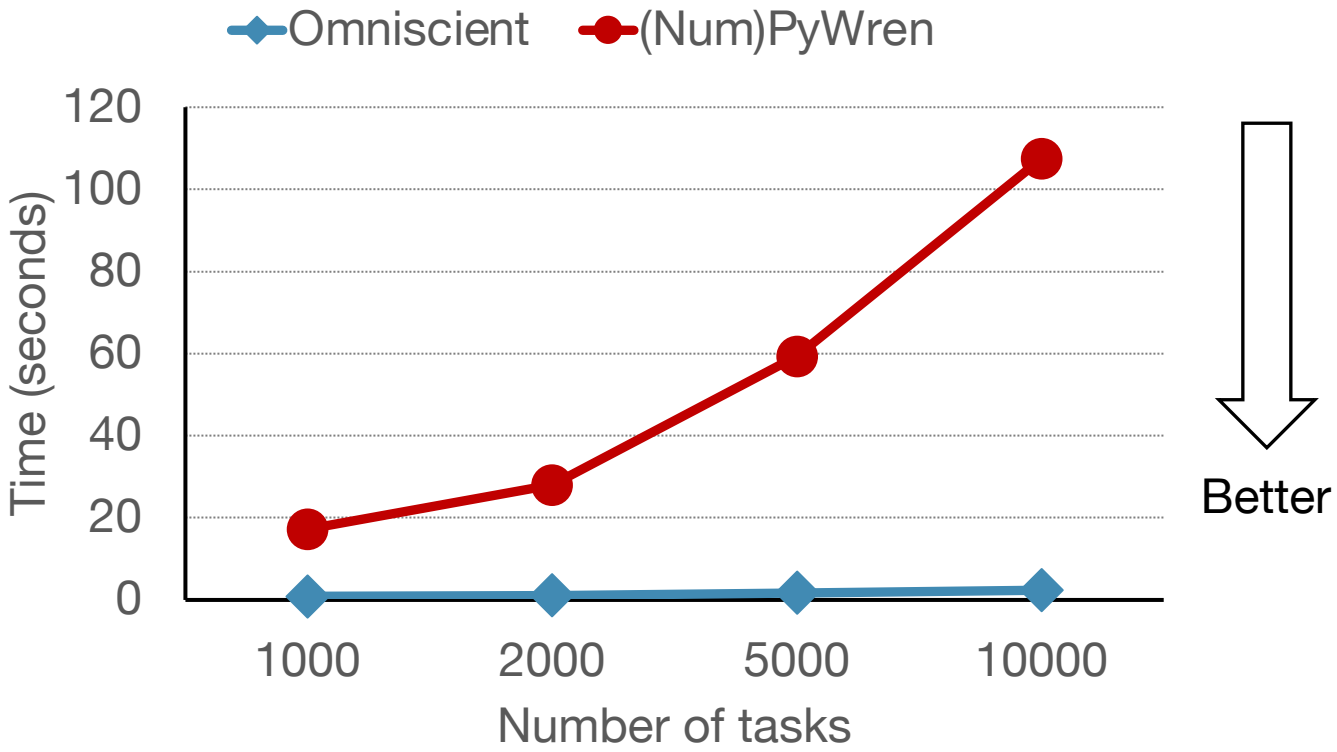


State-of-the-art FaaS frameworks

* [PyWren] Occupy the Cloud: Distributed Computing for the 99%. In ACM SoCC'17.
* [numpywren] Serverless linear algebra. In ACM SoCC'20.

Python analytics on FaaS is too slow!



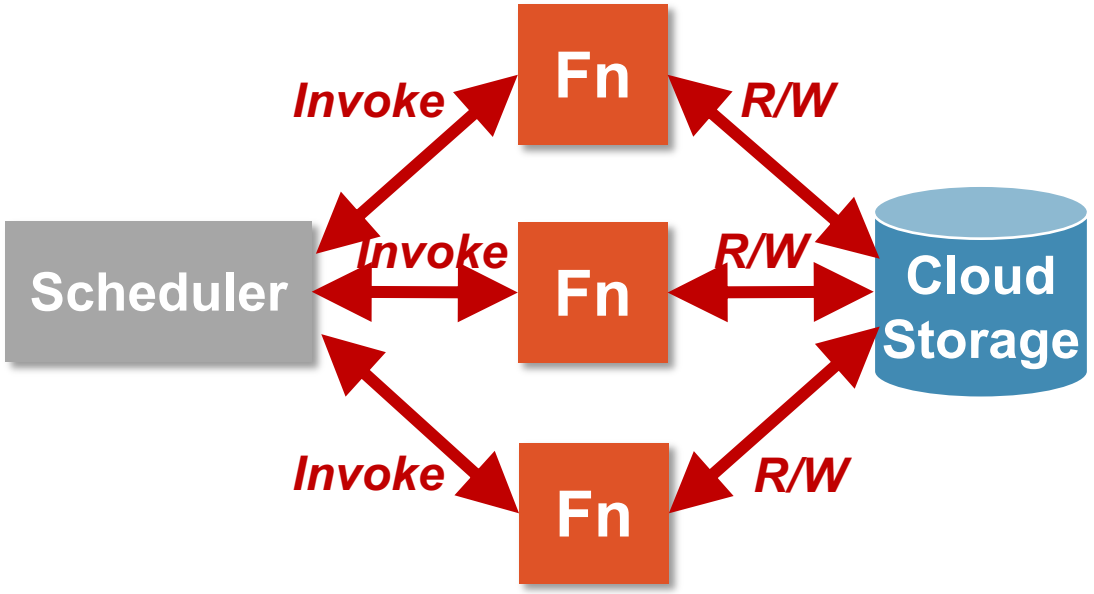


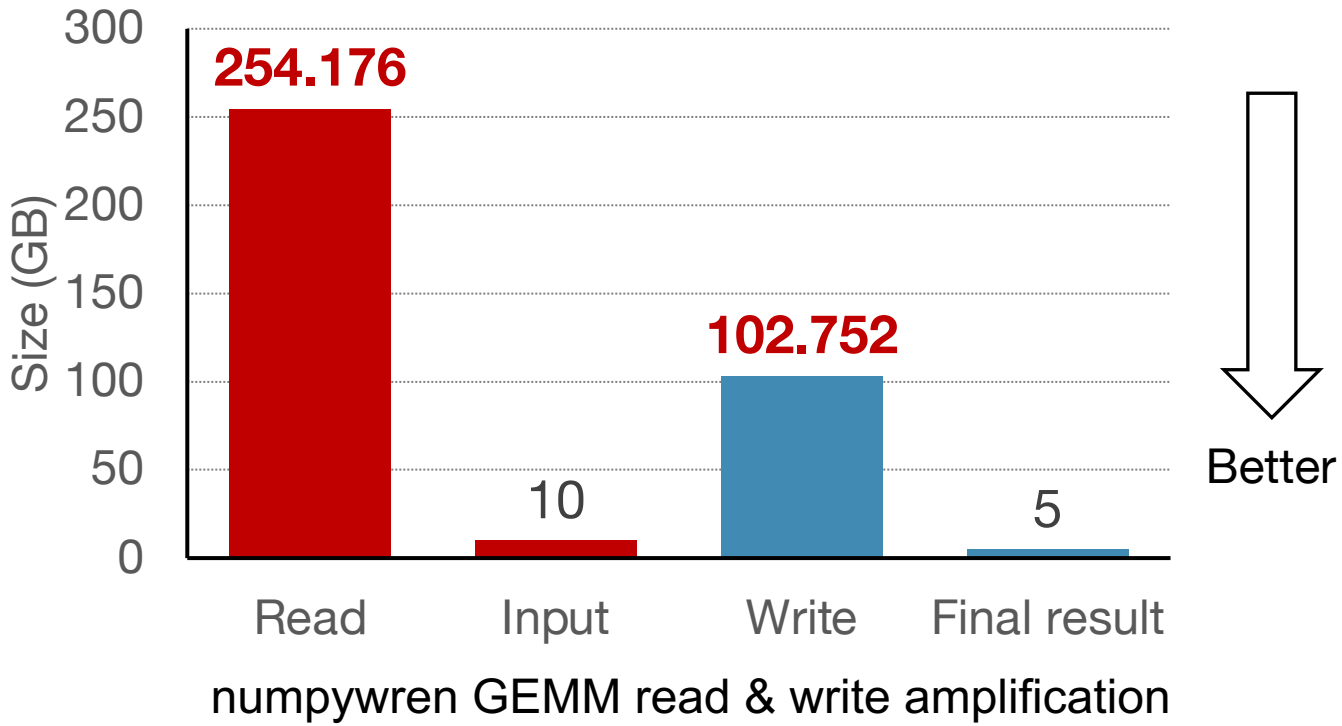
State-of-the-art FaaS frameworks pay huge amounts of FaaS taxes

- **Task scheduling bottleneck:** Too slow to scale to thousands of functions

* [PyWren] Occupy the Cloud: Distributed Computing for the 99%. In ACM SoCC'17.
* [numpywren] Serverless linear algebra. In ACM SoCC'20.

Python analytics on FaaS is too slow!





numpywren GEMM read & write amplification

State-of-the-art FaaS frameworks pay huge amounts of FaaS taxes

- **Task scheduling bottleneck:** Too slow to scale to thousands of functions
- **I/O bottleneck:** Excessive data movement cost due to FaaS constraint

* [PyWren] Occupy the Cloud: Distributed Computing for the 99%. In ACM SoCC'17.
* [numpywren] Serverless linear algebra. In ACM SoCC'20.

Python analytics on FaaS is too slow!

Naively porting a stateful application to a FaaS platform won't work!

Think like a function: A FaaS-centric approach

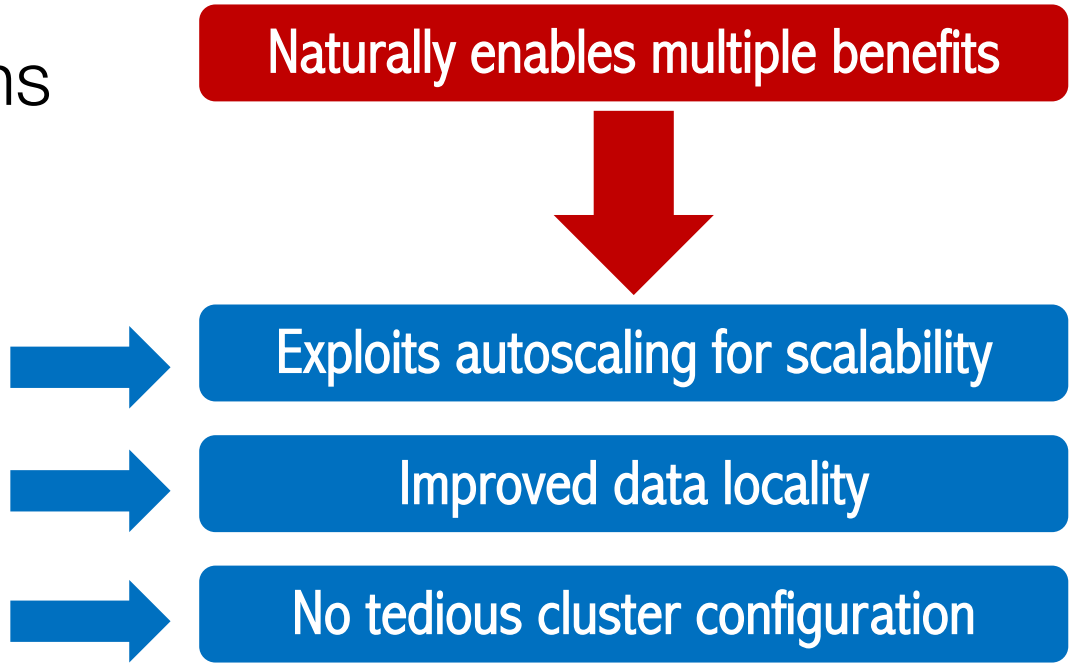
Insight: A FaaS framework may not care about traditional metrics (load balancing, cluster util.)

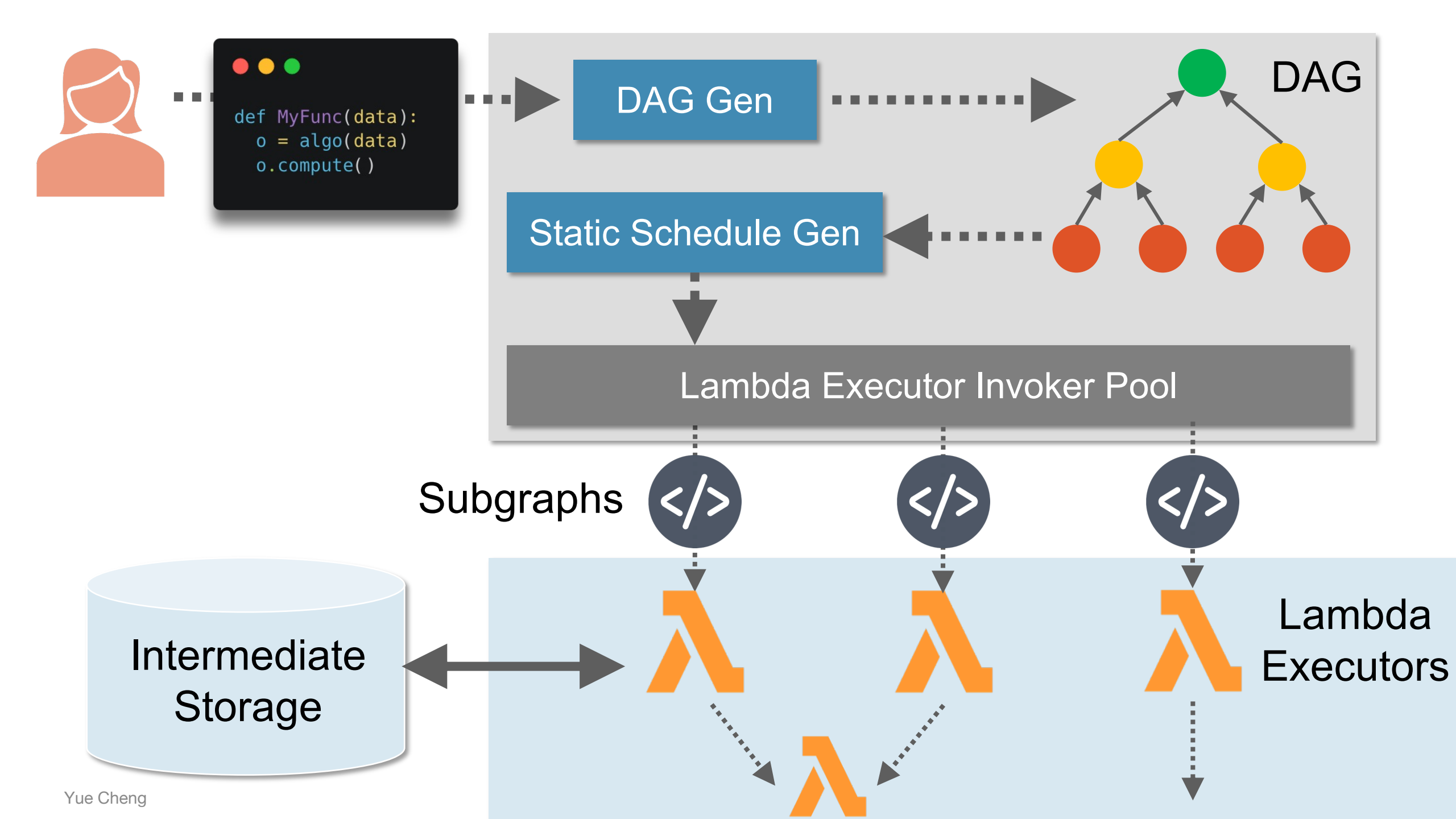
Enter Wukong

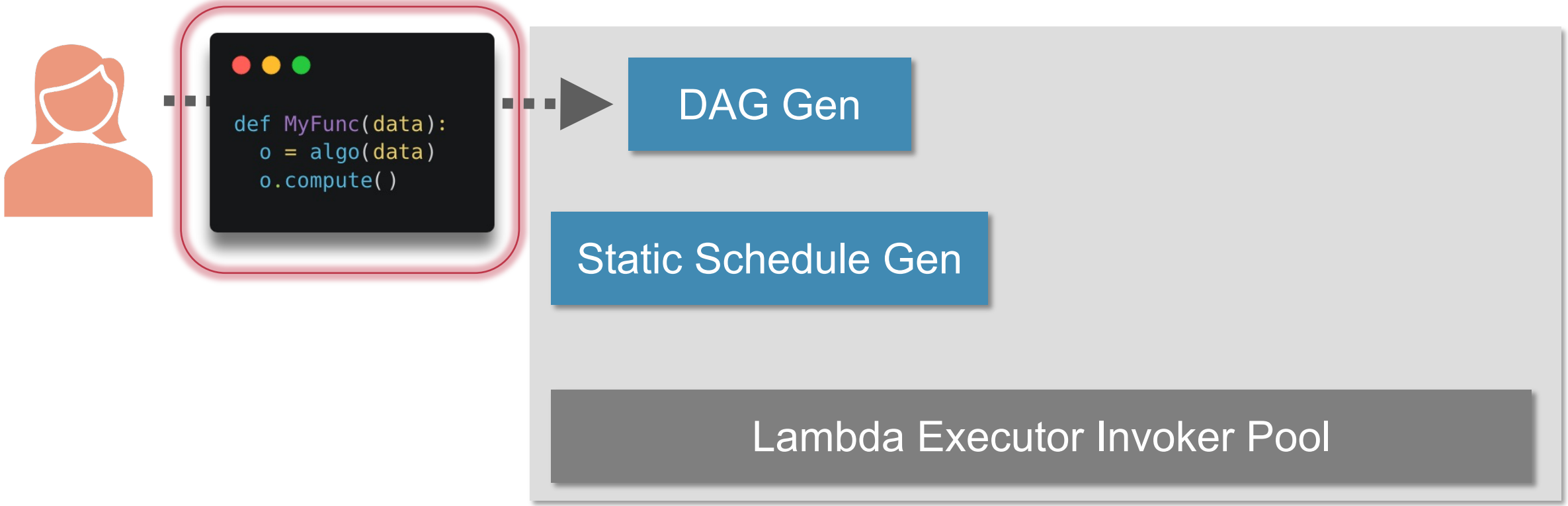
Wukong is a **FaaS-centric** parallel computing framework

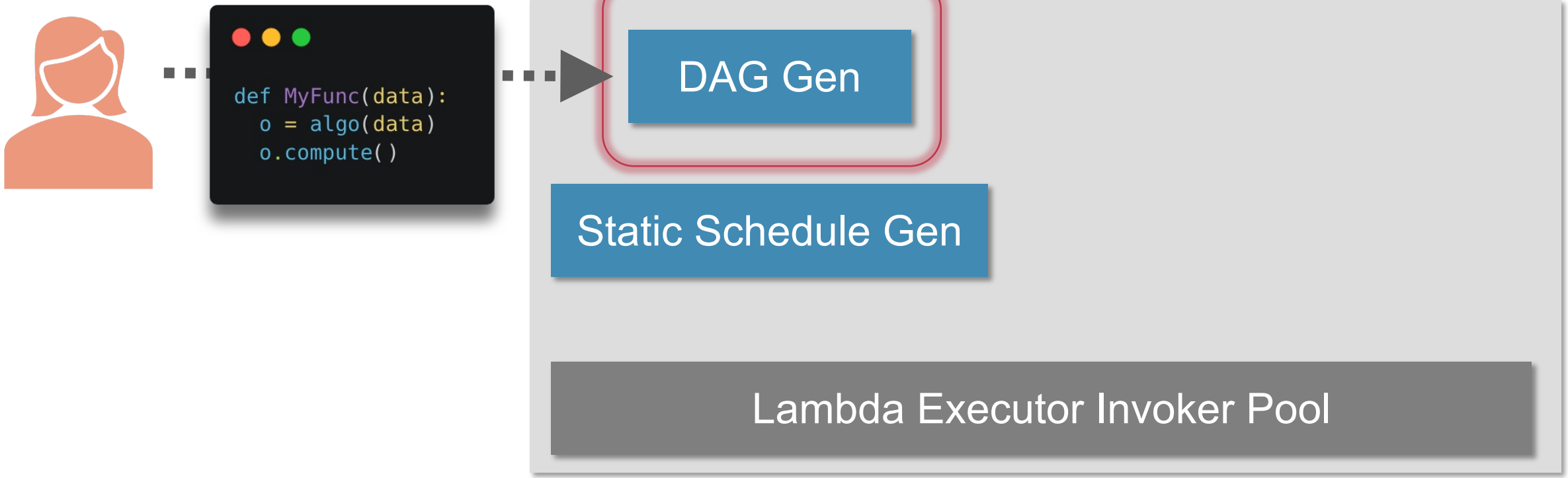
Key idea: Partitions the work of a centralized scheduler across many functions to take advantage of FaaS elasticity

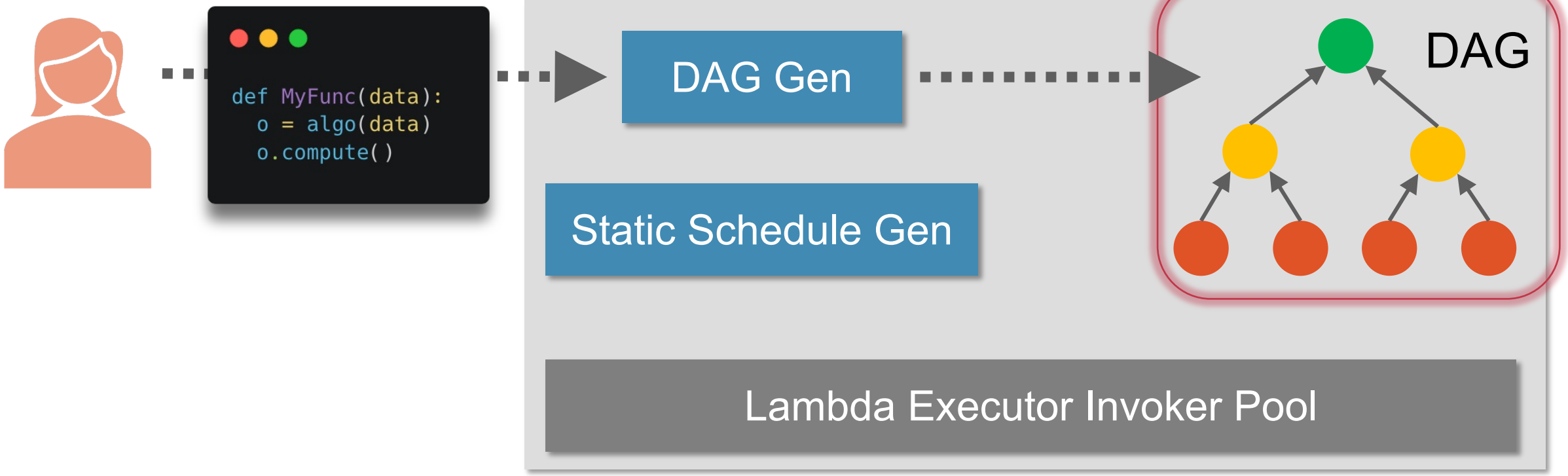
- Functions schedule tasks by invoking functions
- Functions execute multiple tasks to reduce data movement cost
- Functions scale out / in autonomously

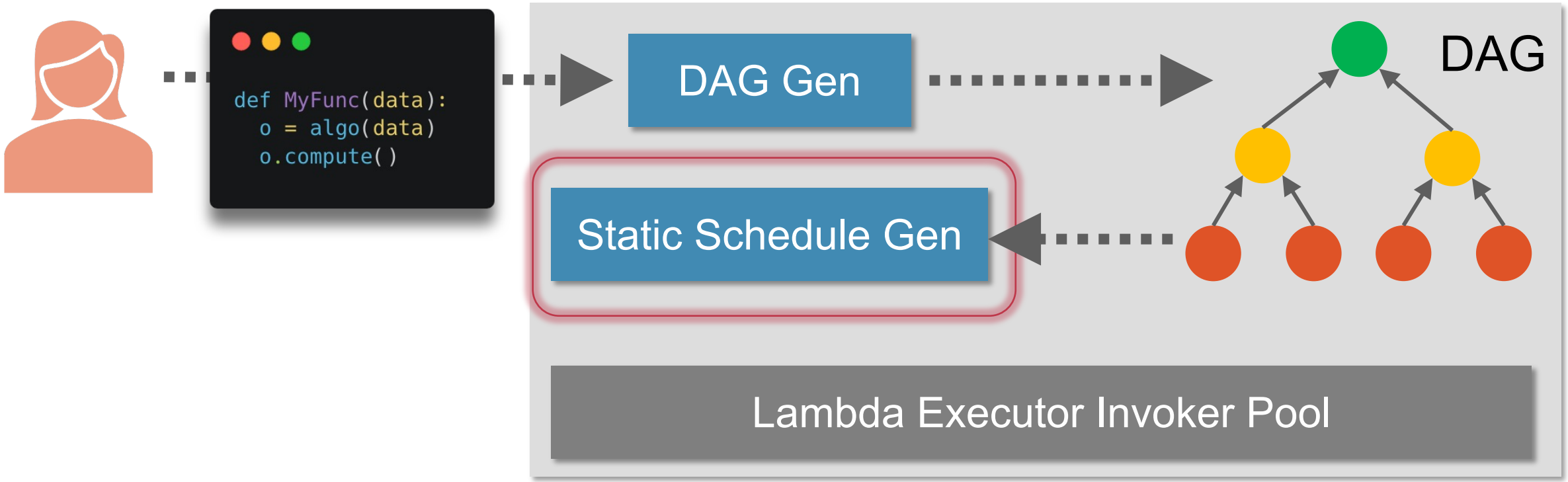


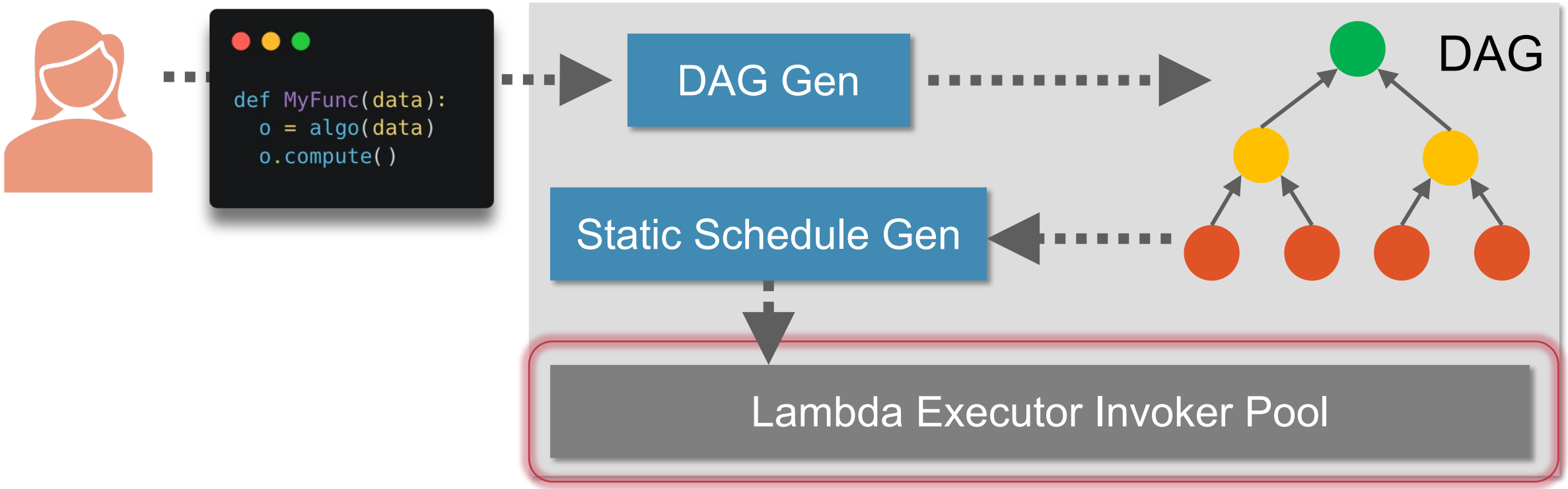

def MyFunc(data):
o = algo(data)
o.compute()
DAG Gen
DAG
Static Schedule Gen
Lambda Executor Invoker Pool
Subgraphs
Intermediate Storage
Lambda Executors

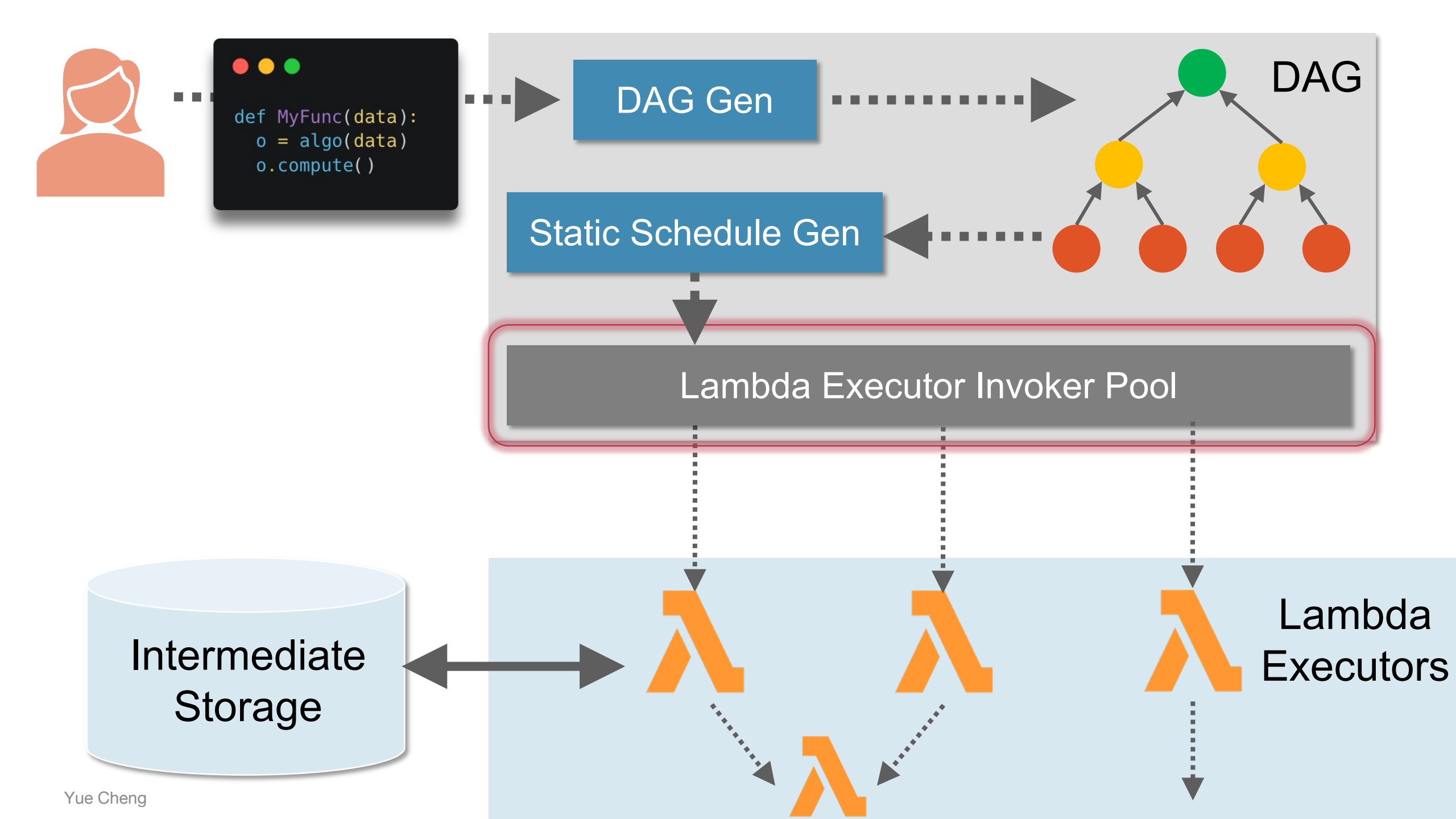

def MyFunc(data):
o = algo(data)
o.compute()
DAG Gen
Static Schedule Gen
Lambda Executor Invoker Pool

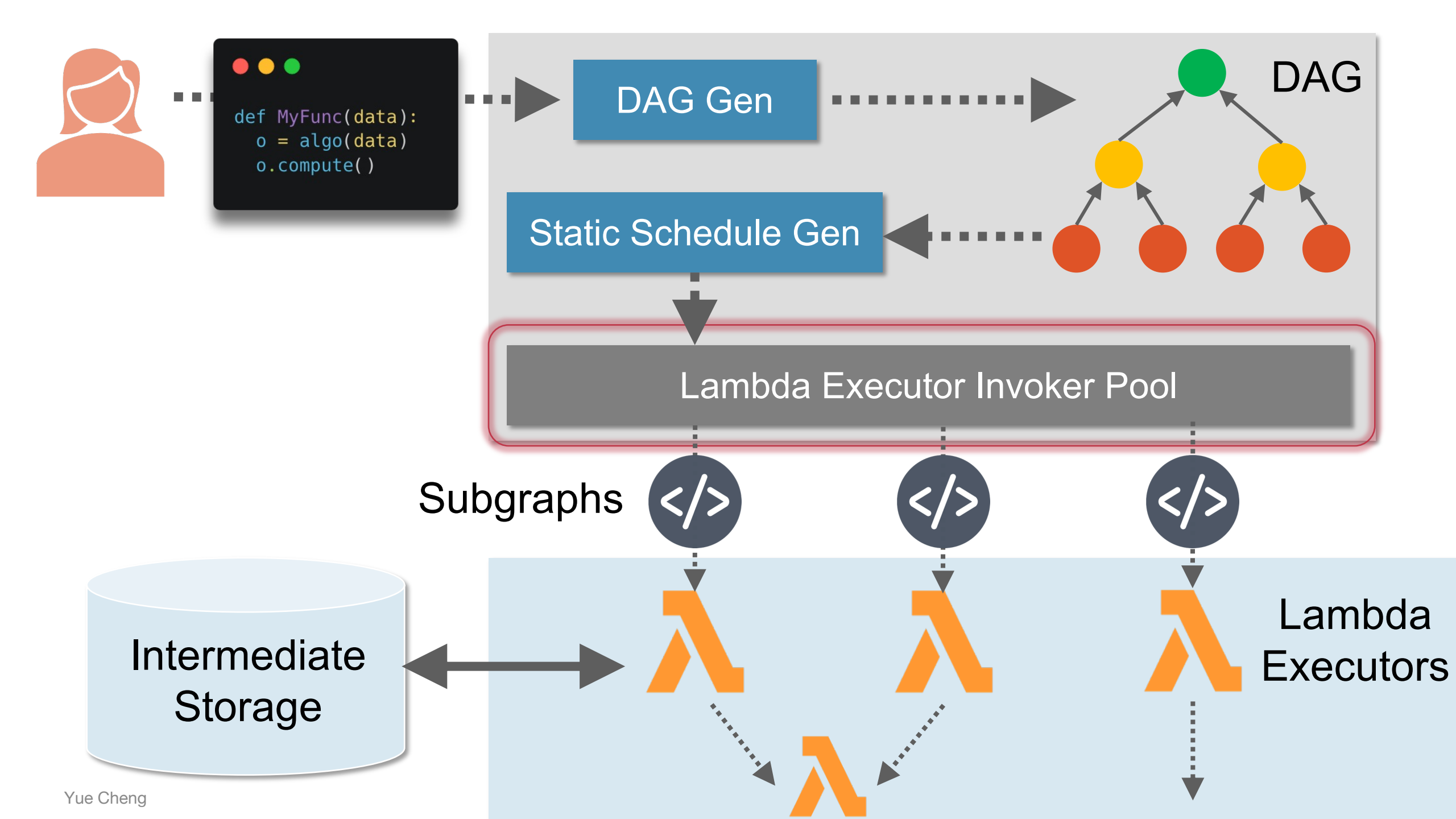

def MyFunc(data):
o = algo(data)
o.compute()
DAG Gen
Static Schedule Gen
Lambda Executor Invoker Pool

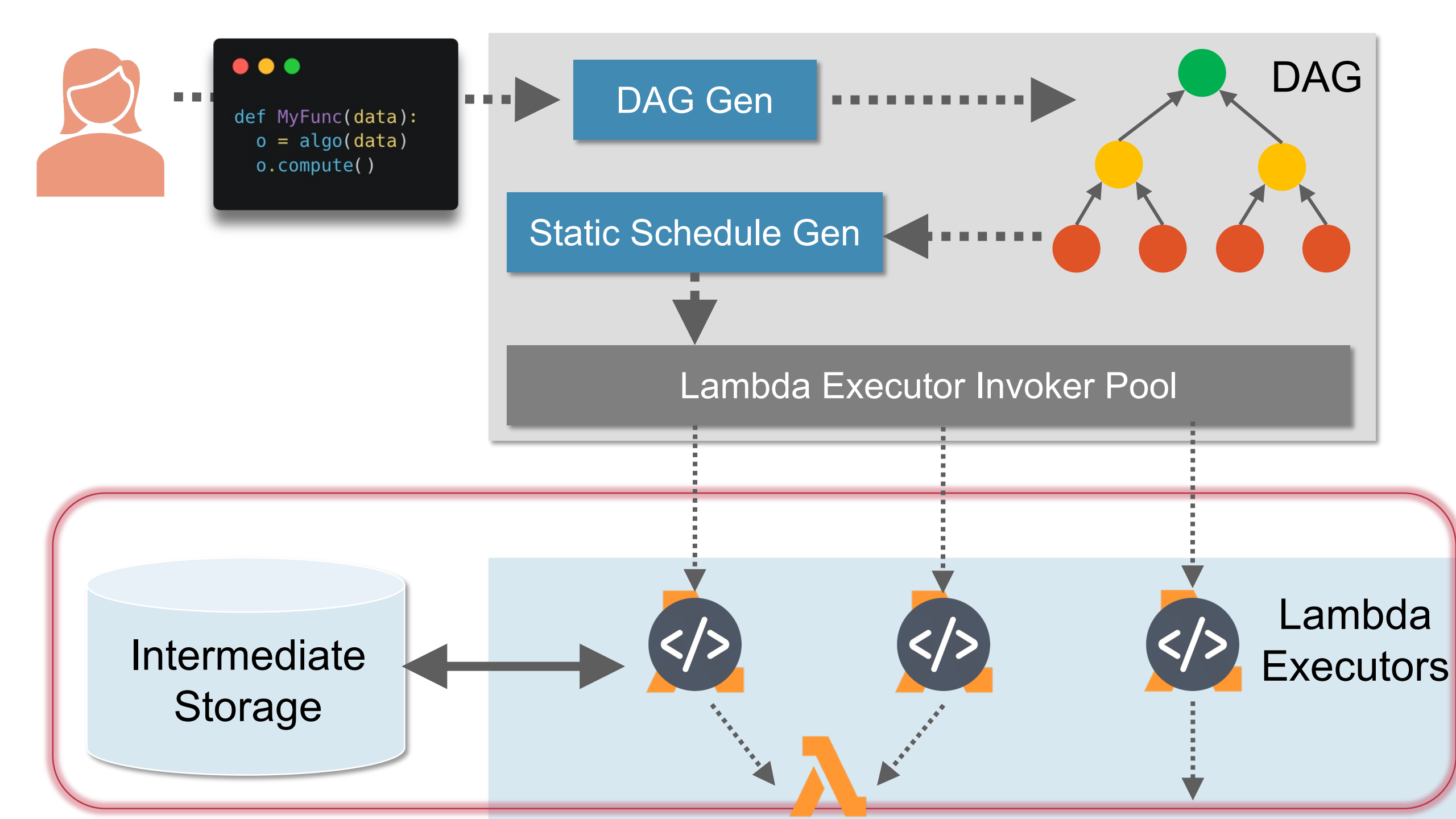

def MyFunc(data):
o = algo(data)
o.compute()
DAG Gen
DAG
Static Schedule Gen
Lambda Executor Invoker Pool


def MyFunc(data):
o = algo(data)
o.compute()
DAG Gen
DAG
Static Schedule Gen
Lambda Executor Invoker Pool


def MyFunc(data):
o = algo(data)
o.compute()
DAG Gen
DAG
Static Schedule Gen
Lambda Executor Invoker Pool

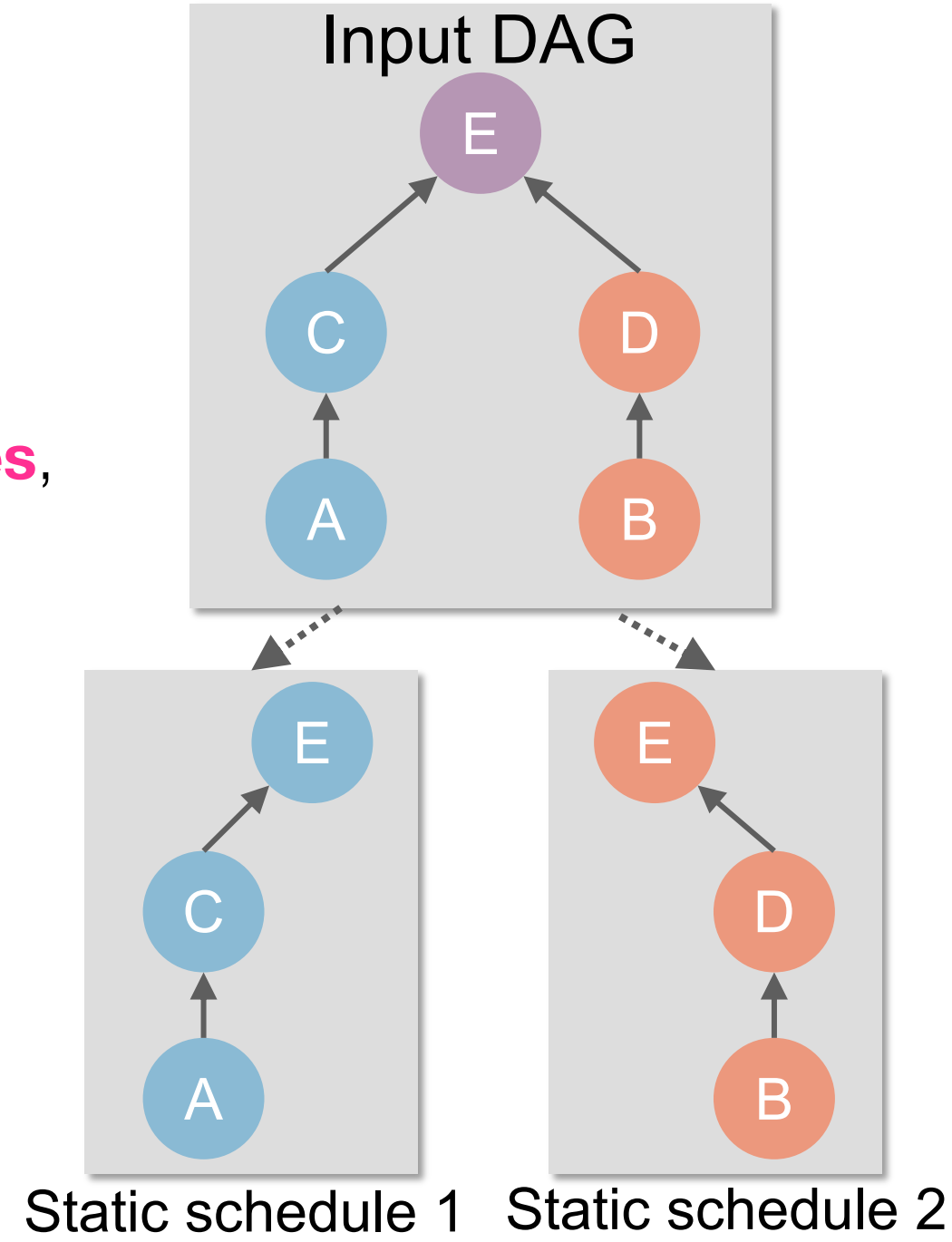

def MyFunc(data):
o = algo(data)
o.compute()
DAG Gen
DAG
Static Schedule Gen
Lambda Executor Invoker Pool
Intermediate Storage
Lambda Executors


def MyFunc(data):
o = algo(data)
o.compute()
DAG Gen
DAG
Static Schedule Gen
Lambda Executor Invoker Pool
Subgraphs
Intermediate Storage
Lambda Executors


def MyFunc(data):
o = algo(data)
o.compute()
DAG Gen
DAG
Static Schedule Gen
Lambda Executor Invoker Pool
Intermediate Storage
Lambda Executors

Scheduling in Wukong

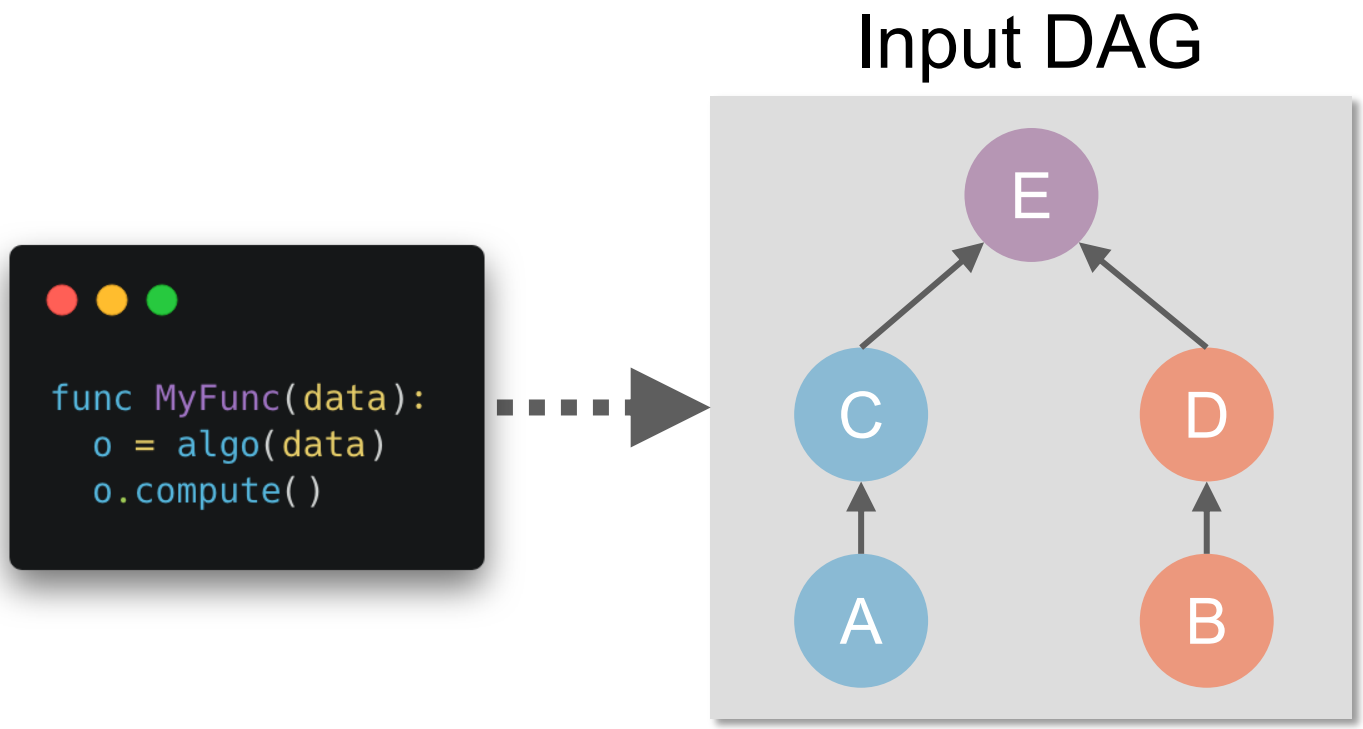
- Combination of **static** and **dynamic** scheduling
- Input DAG partitioned into **static schedules**, or subgraphs of the original DAG
- Serverless executors are assigned a **static schedule**
- Executors use **dynamic scheduling** to enforce data dependencies and **cooperatively** schedule tasks found in multiple static schedules



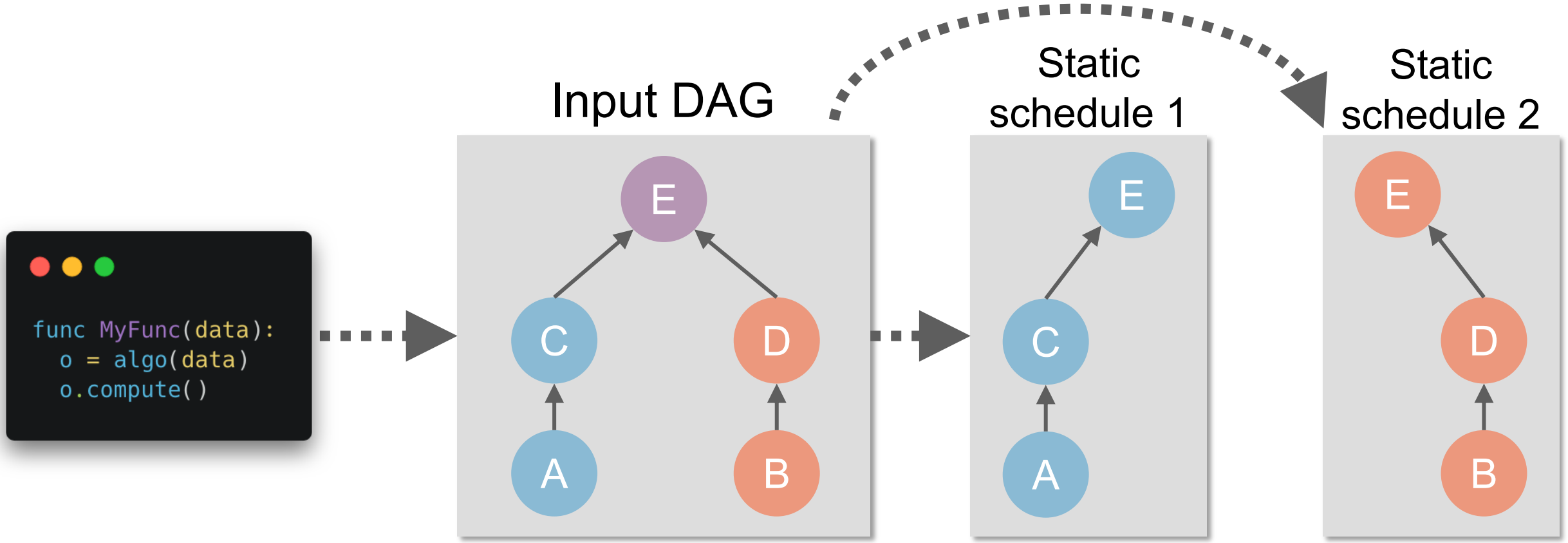
```
func MyFunc(data):
  o = algo(data)
  o.compute()
```

Static scheduling

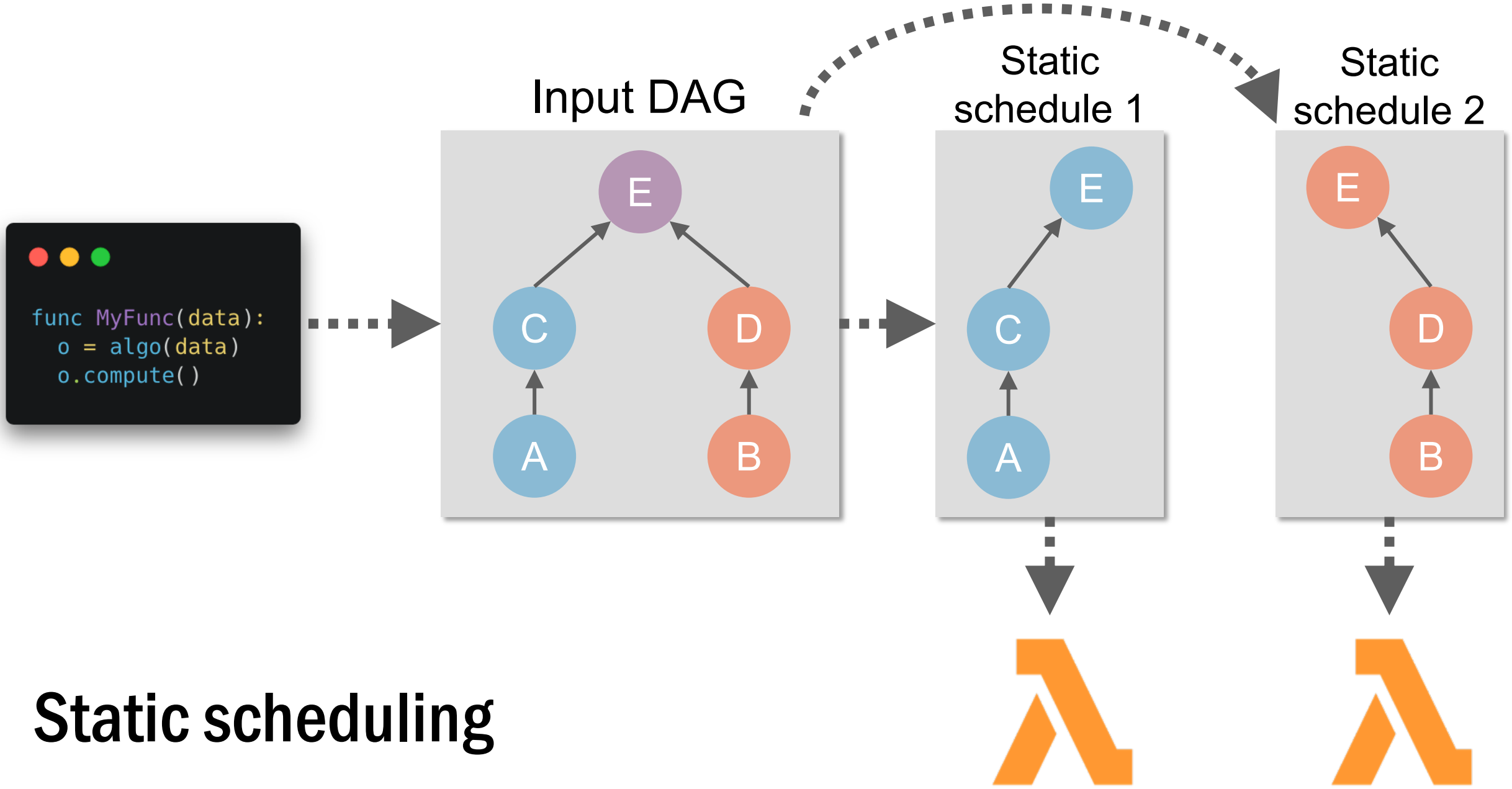
Static scheduling



Static scheduling

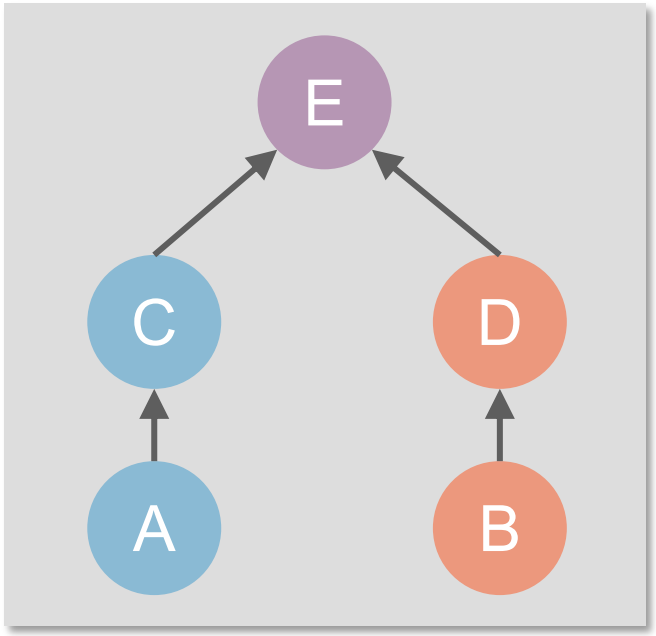


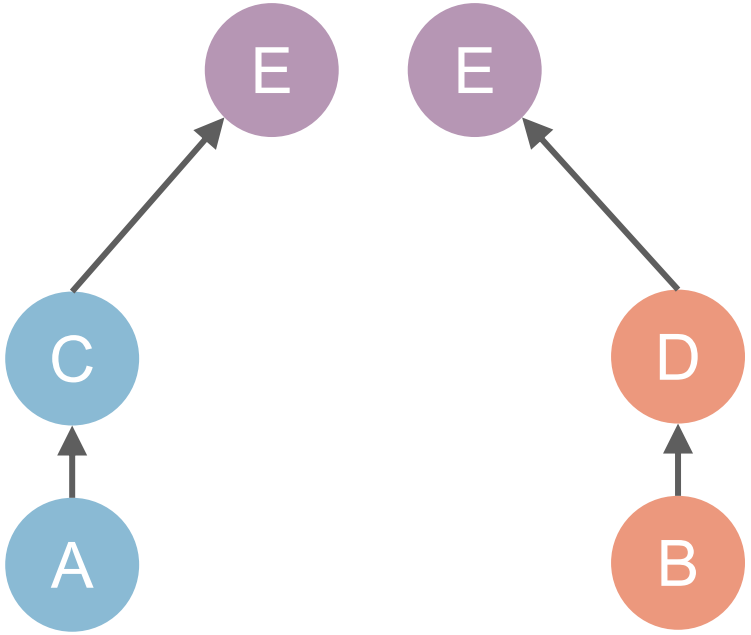
Static scheduling



Static scheduling

Input DAG



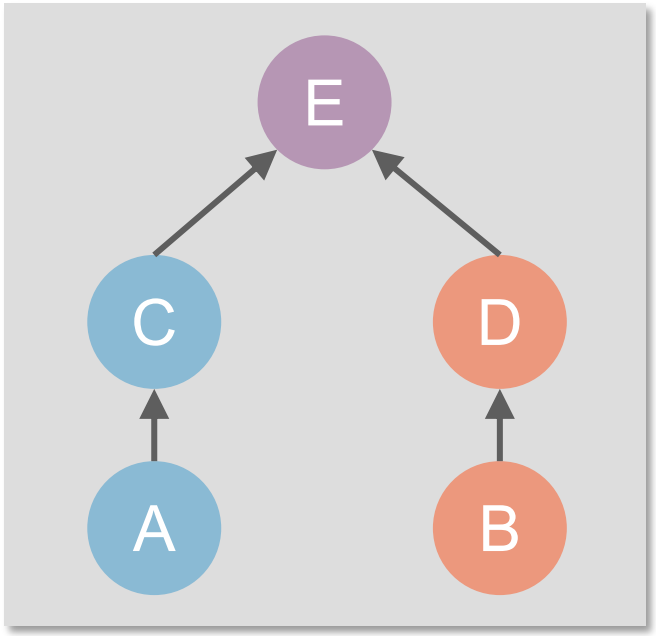


Dynamic scheduling

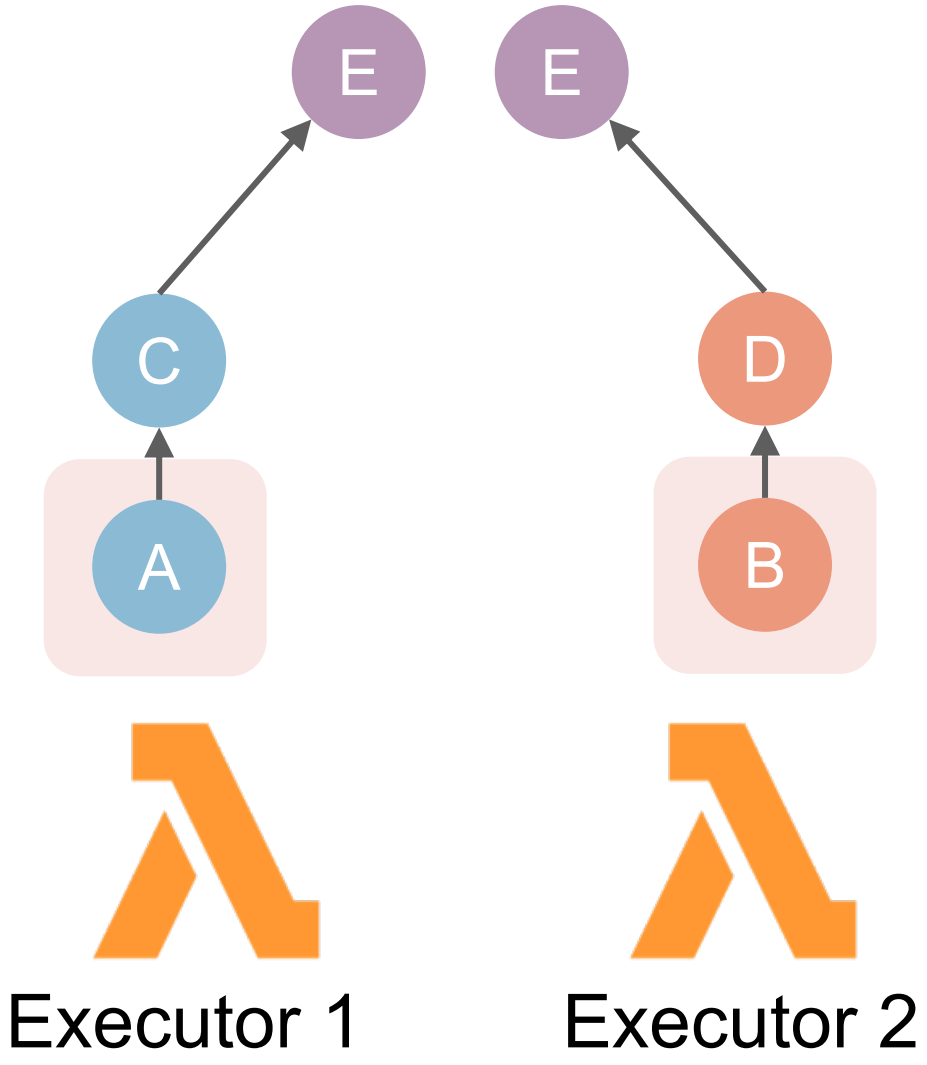
Executor 1

Executor 2

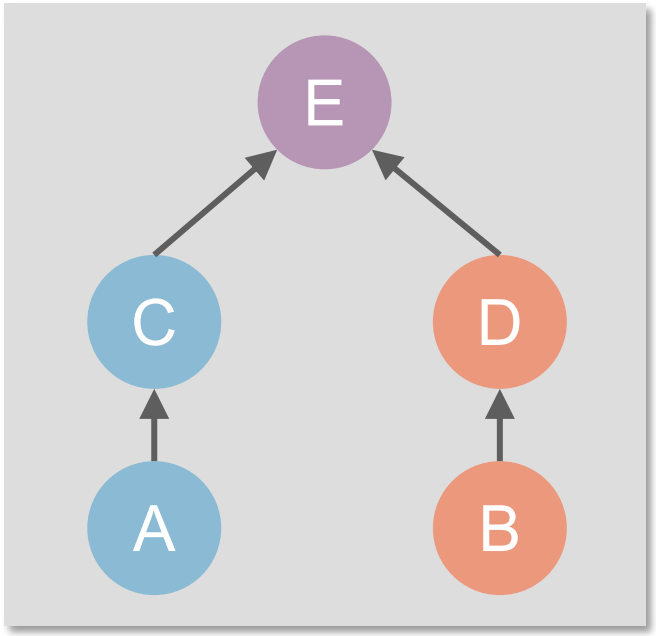
Input DAG

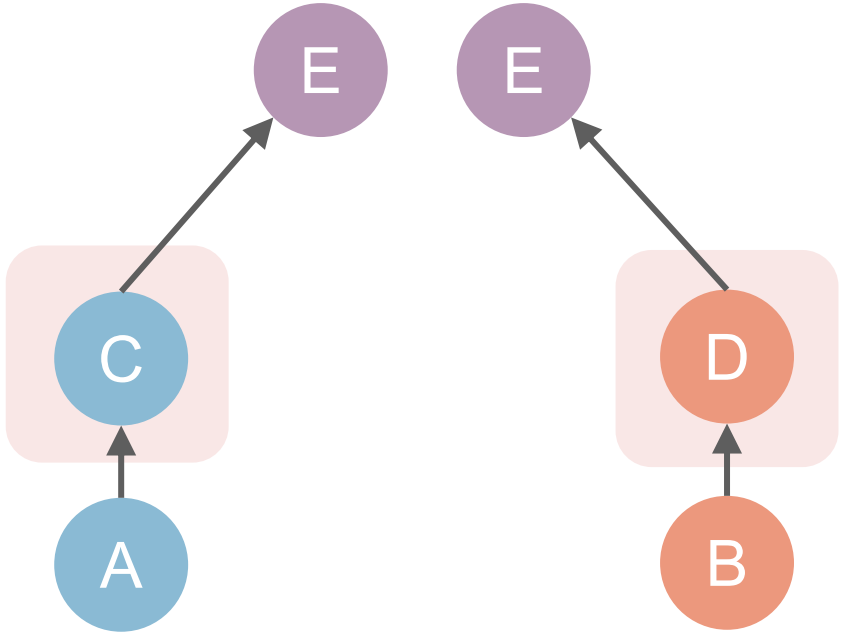


Dynamic scheduling



Input DAG



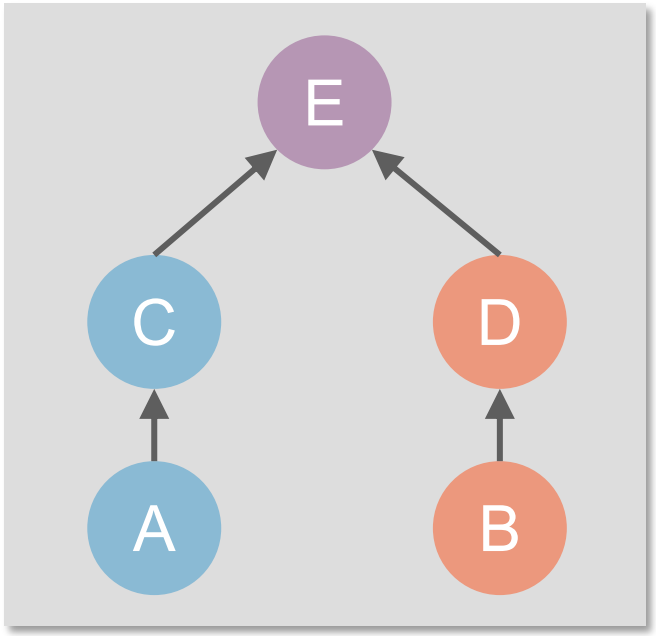


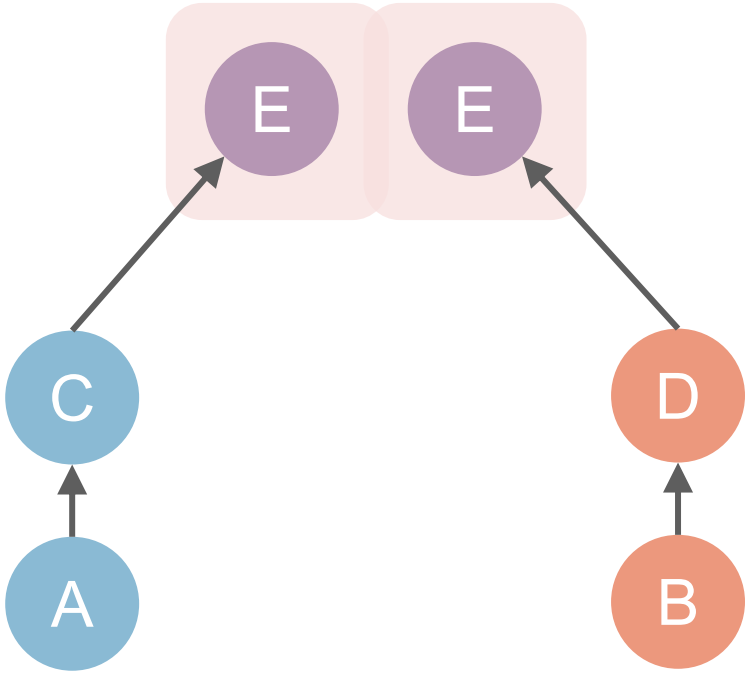
Dynamic scheduling

Executor 1

Executor 2

Input DAG



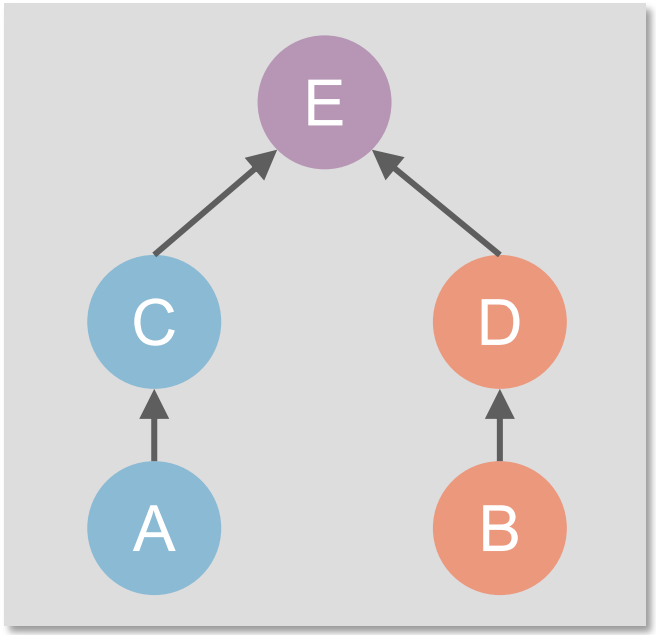


Dynamic scheduling

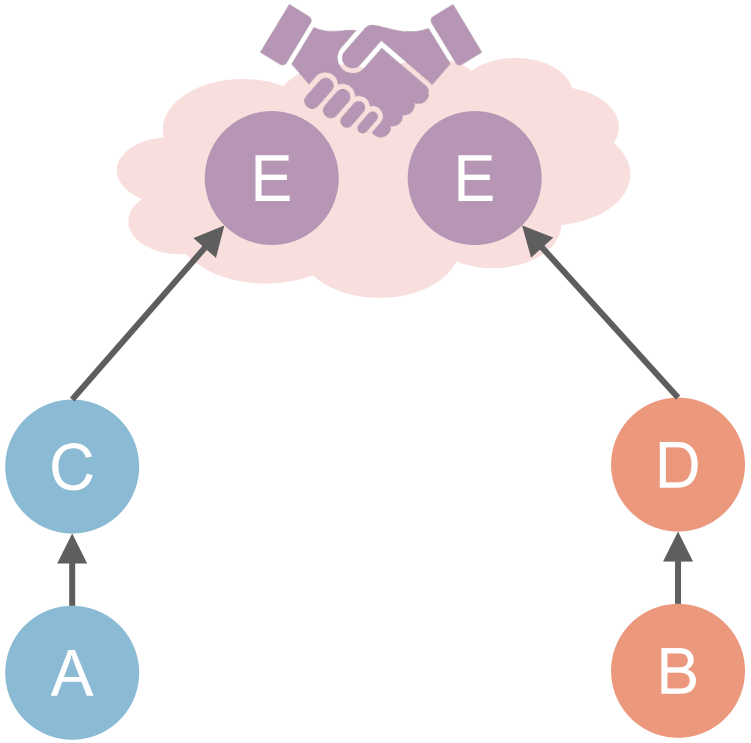
Executor 1

Executor 2

Input DAG



Cooperate

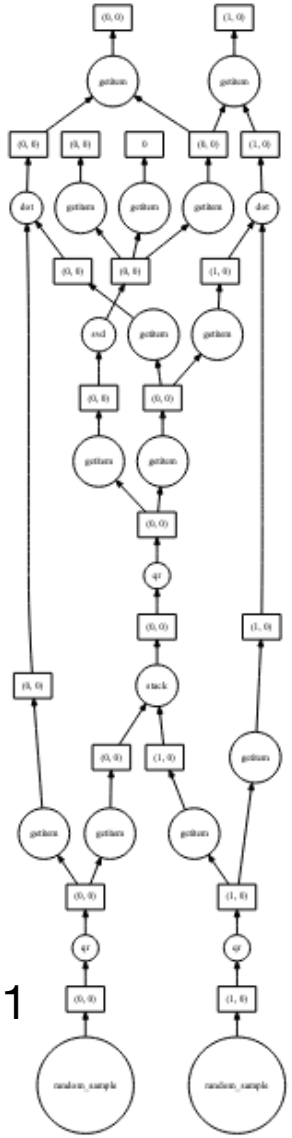


Dynamic scheduling

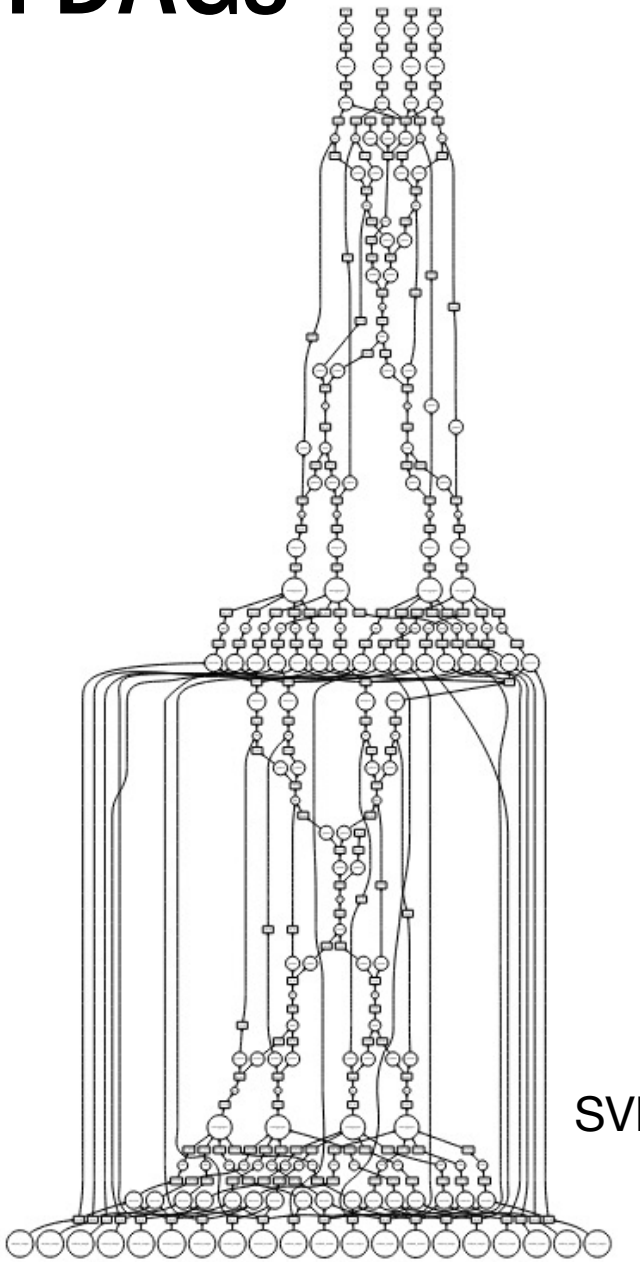
Executor 1

Executor 2

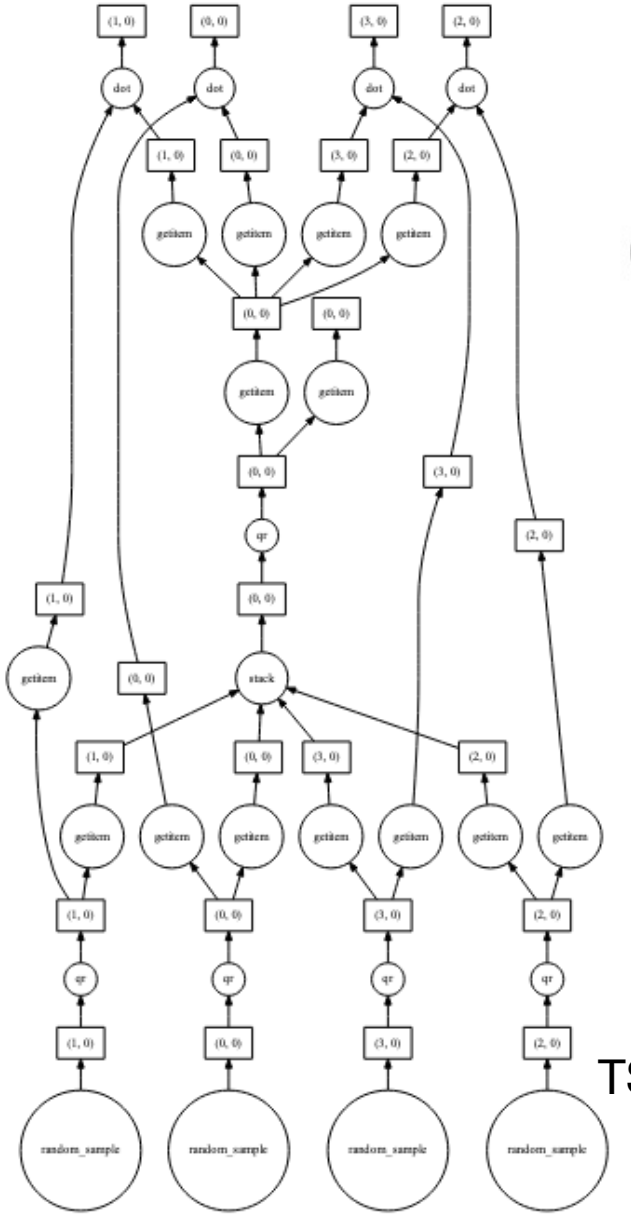
Application DAGs



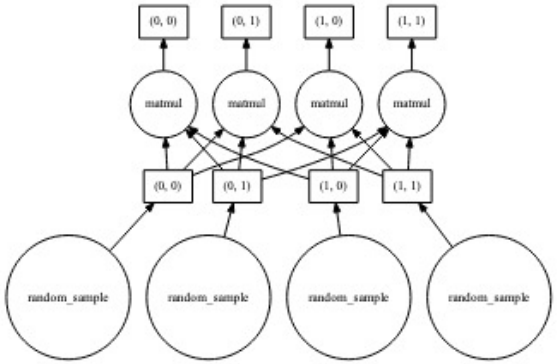
SVD1



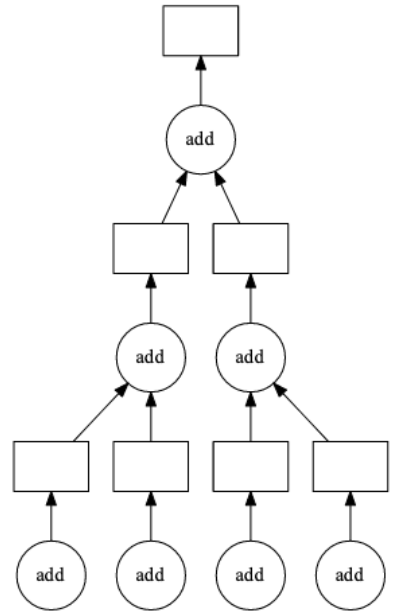
SVD2



TSQR

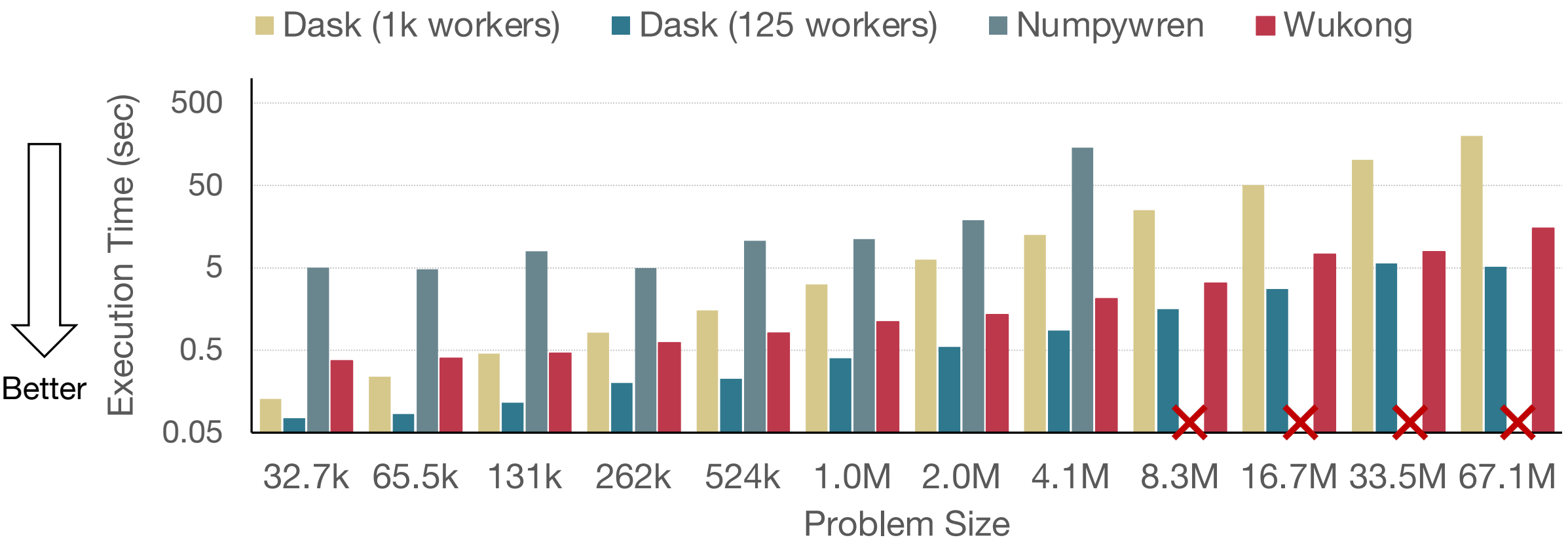


GEMM



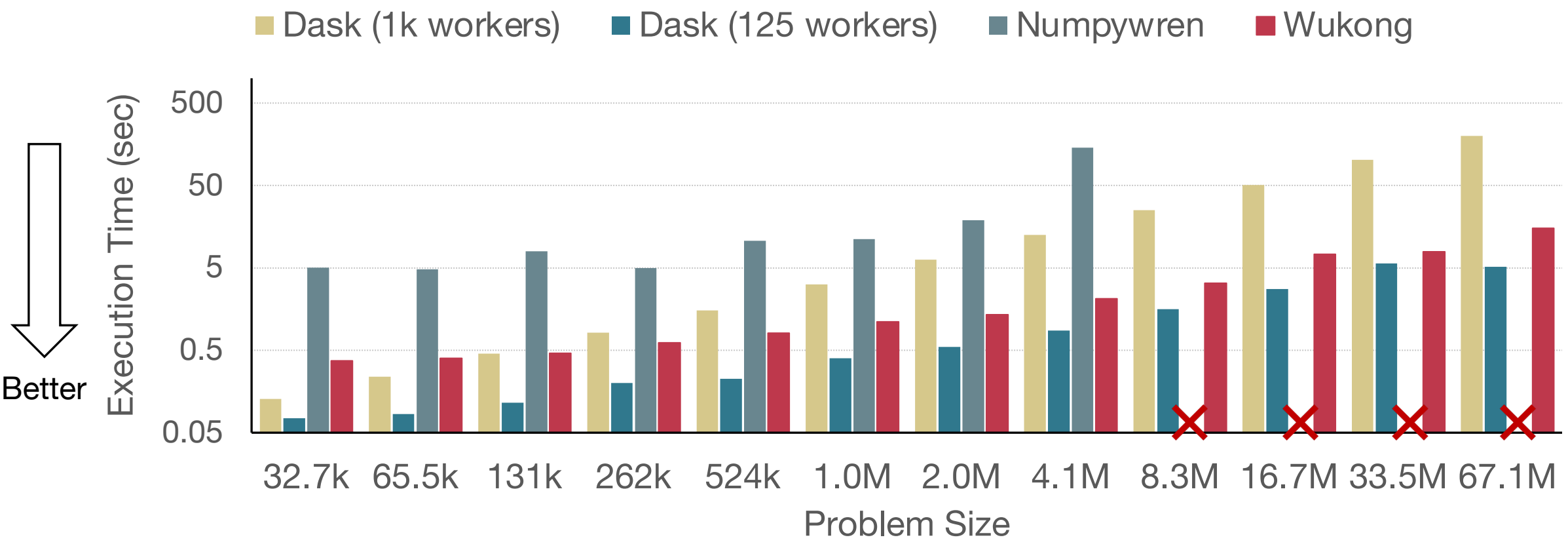
Tree reduction

Application performance: TSQR



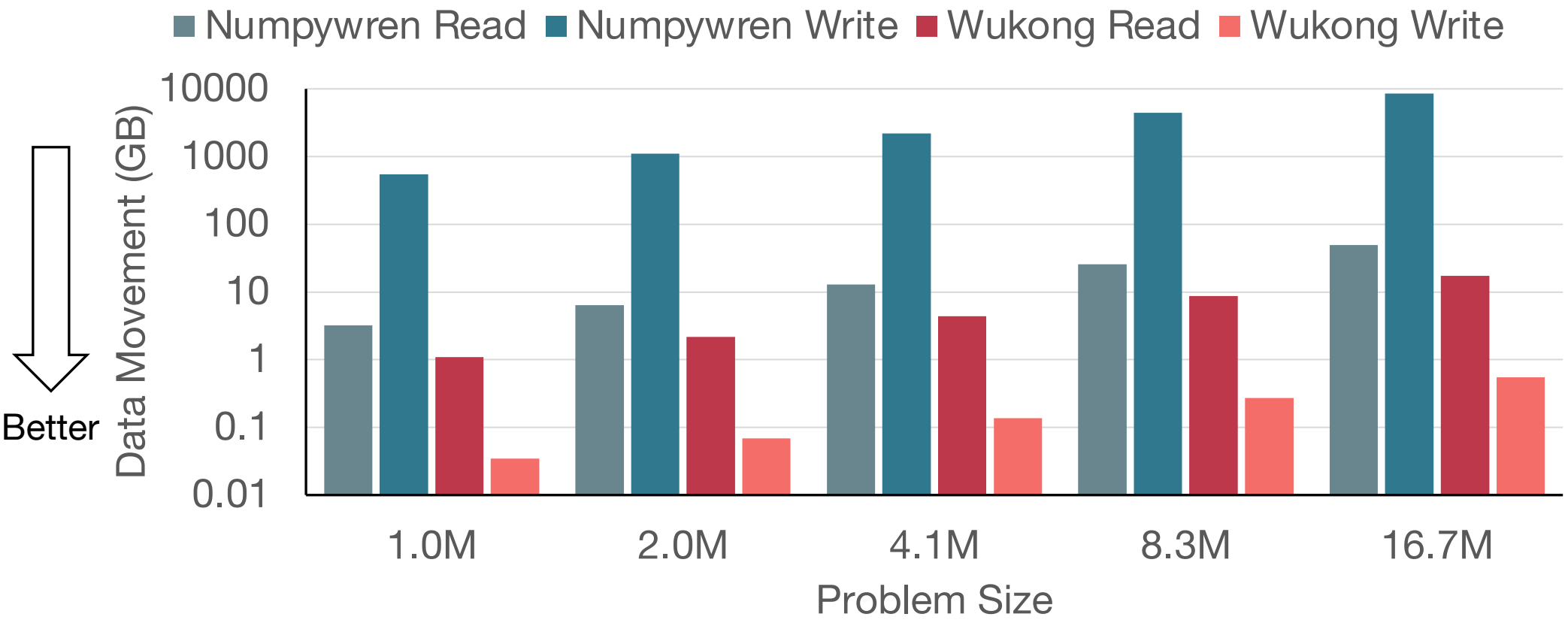
Wukong and numpywren ran on AWS Lambda w/ 3GB memory
Dask distributed ran on 125 c5.4xlarge EC2 VMs w/ 2,000 vCPU cores

Application performance: TSQR

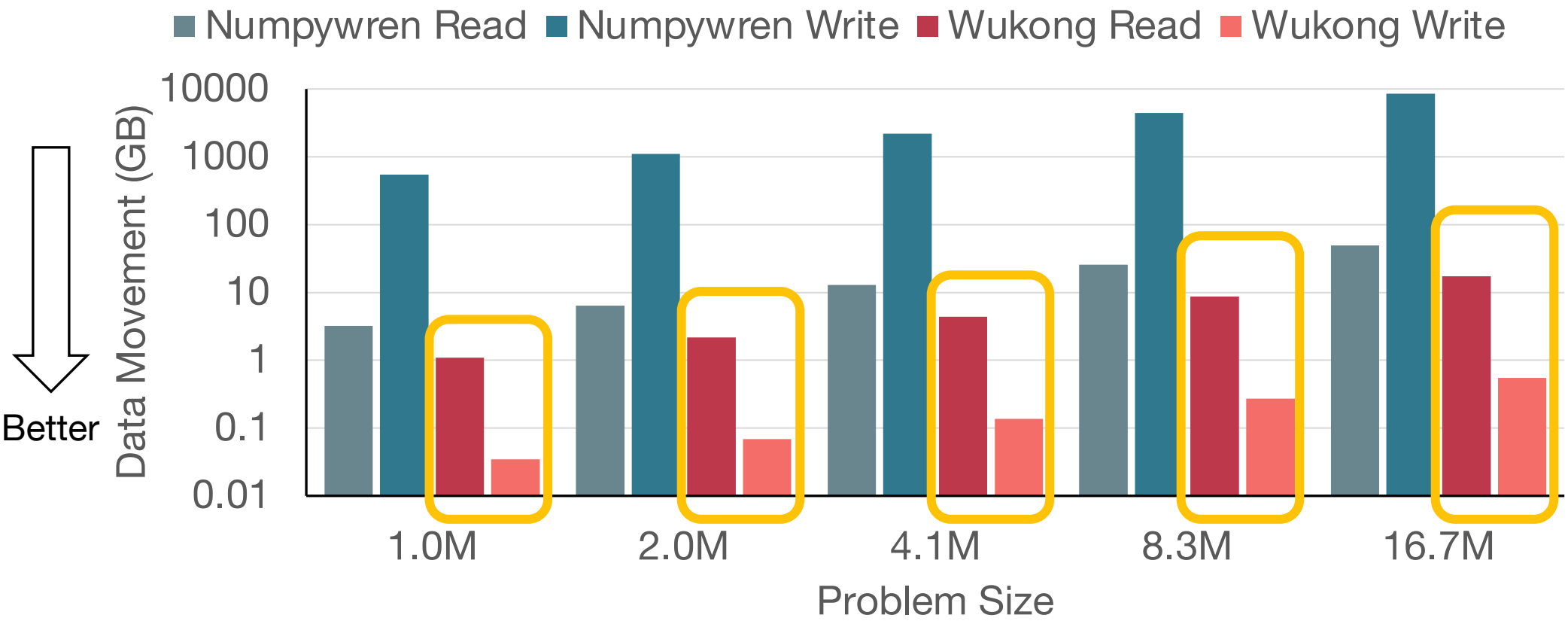


Wukong outperforms numpywren considerably for all problem sizes

Data movement cost: TSQR



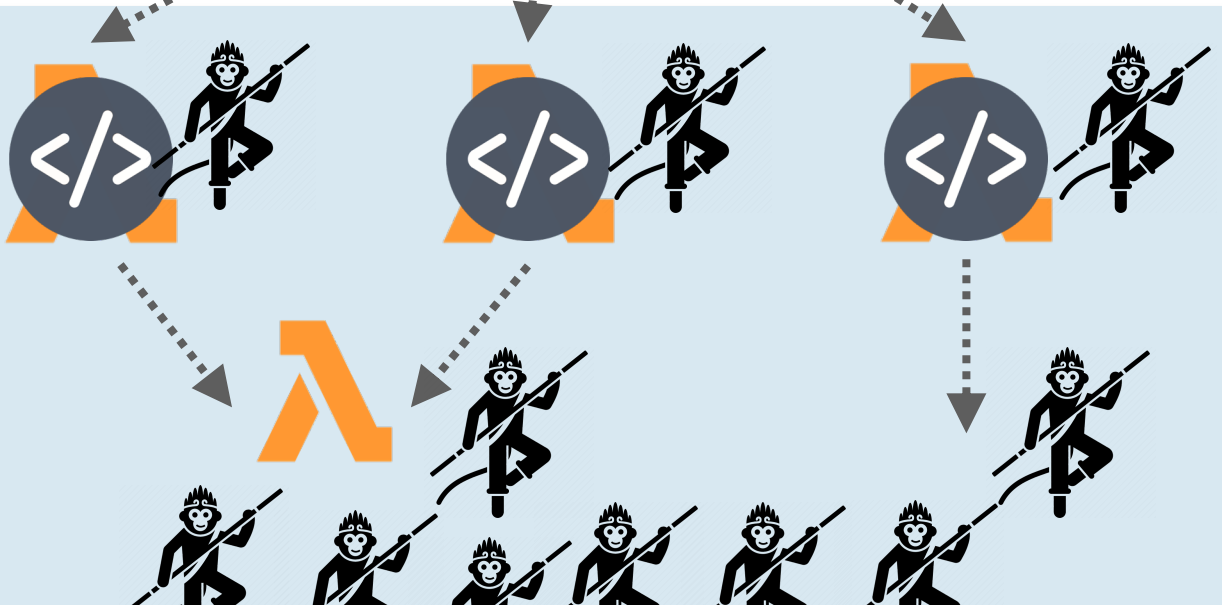
Data movement cost: TSQR



Wukong reads and writes considerably less data than numpywren

Backup slides

Wukong's magic hairs vs. decentralized scheduling



Wukong performance

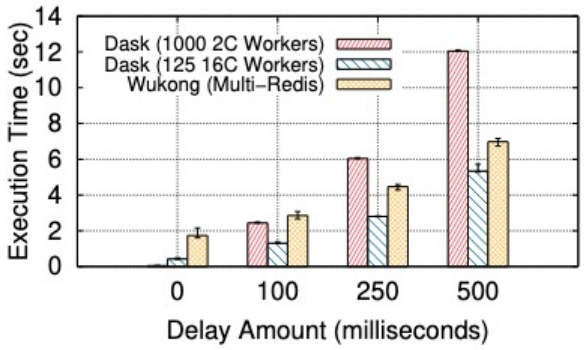


Figure 9: TR.

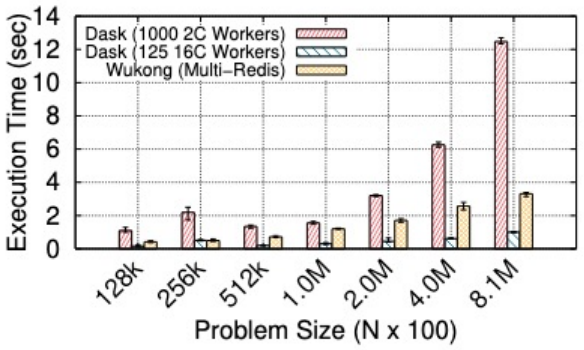


Figure 10: SVD1.

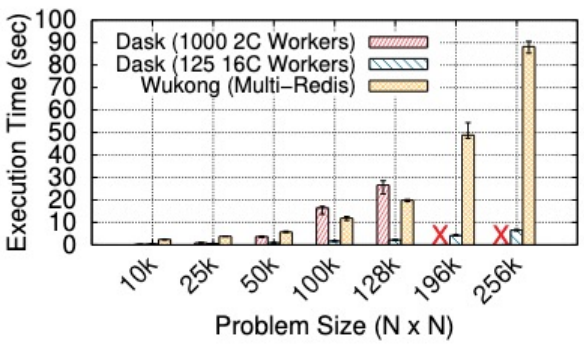


Figure 11: SVD2.

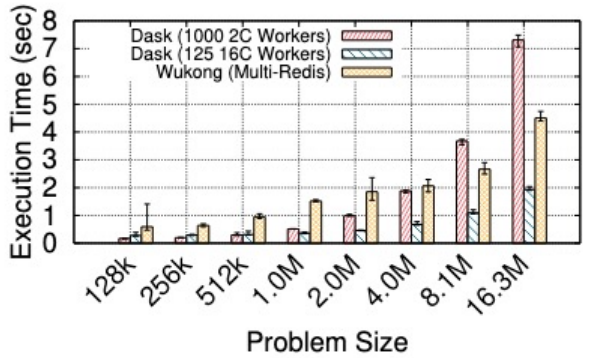


Figure 12: SVC.

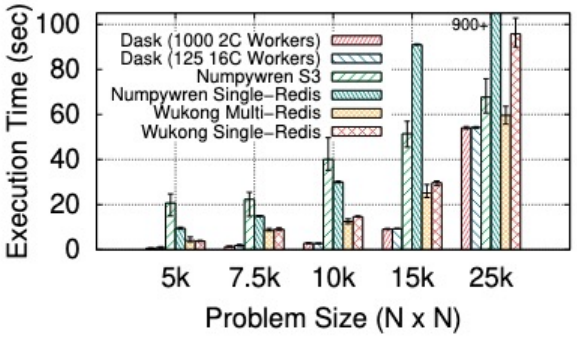


Figure 13: GEMM.

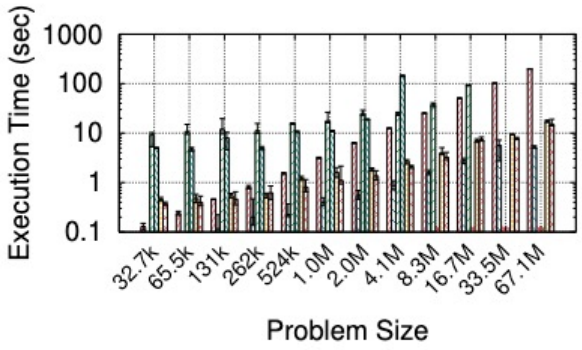


Figure 14: TSQR (log-scale).

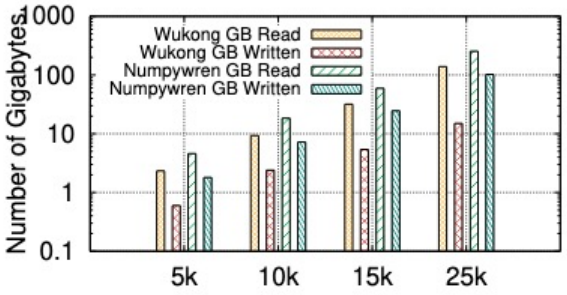


Figure 15: GEMM I/O (log).

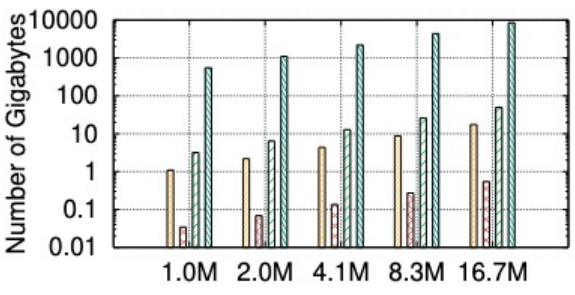
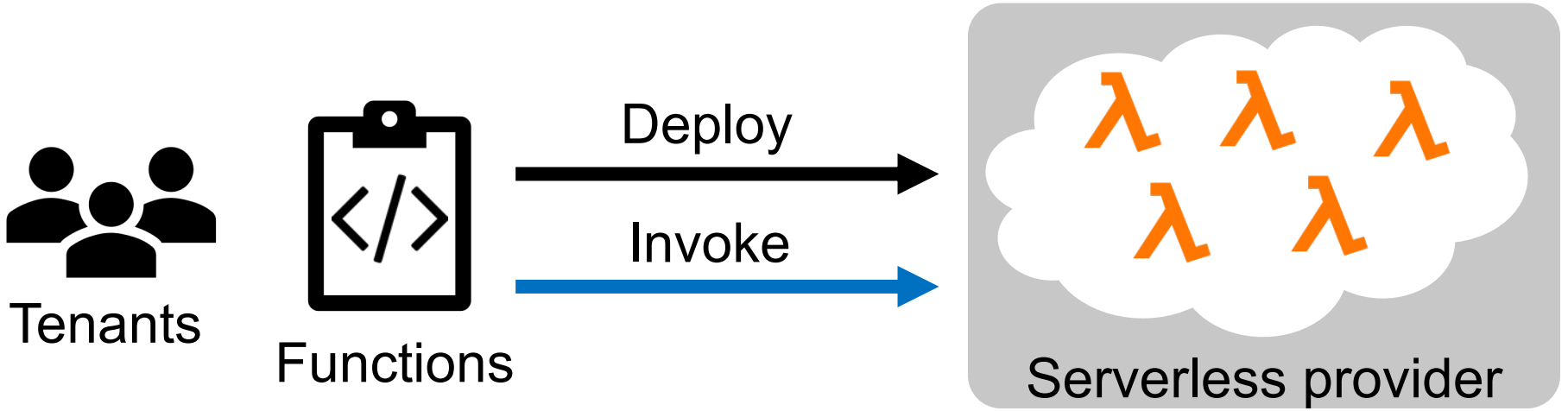


Figure 16: TSQR I/O (log).

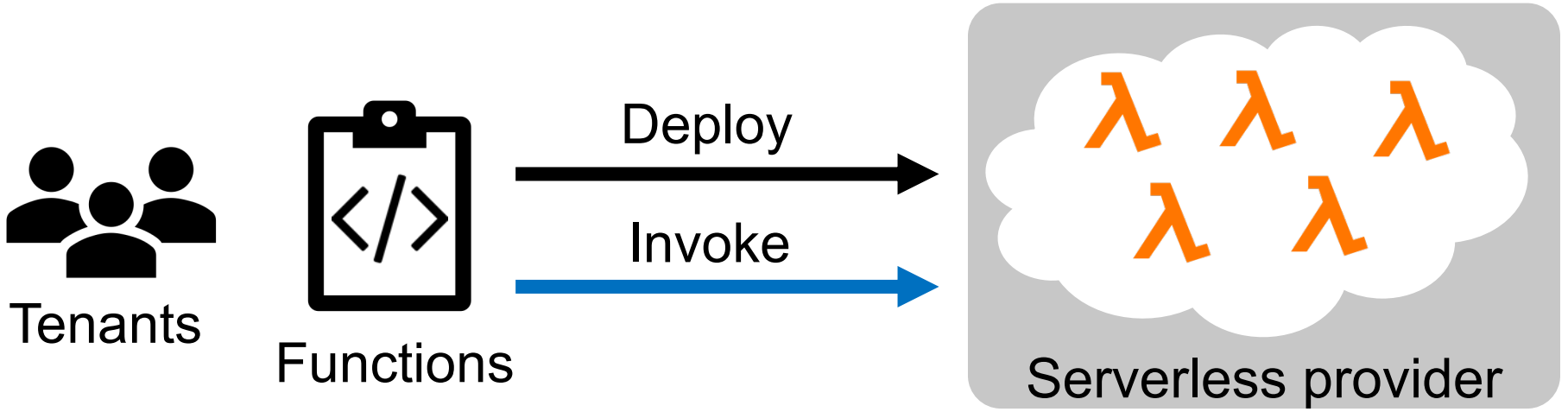
A primer on Serverless Computing

- Serverless computing enables cloud tenants to launch short-lived tasks (i.e., Lambda functions) with high elasticity and fine-grained resource billing



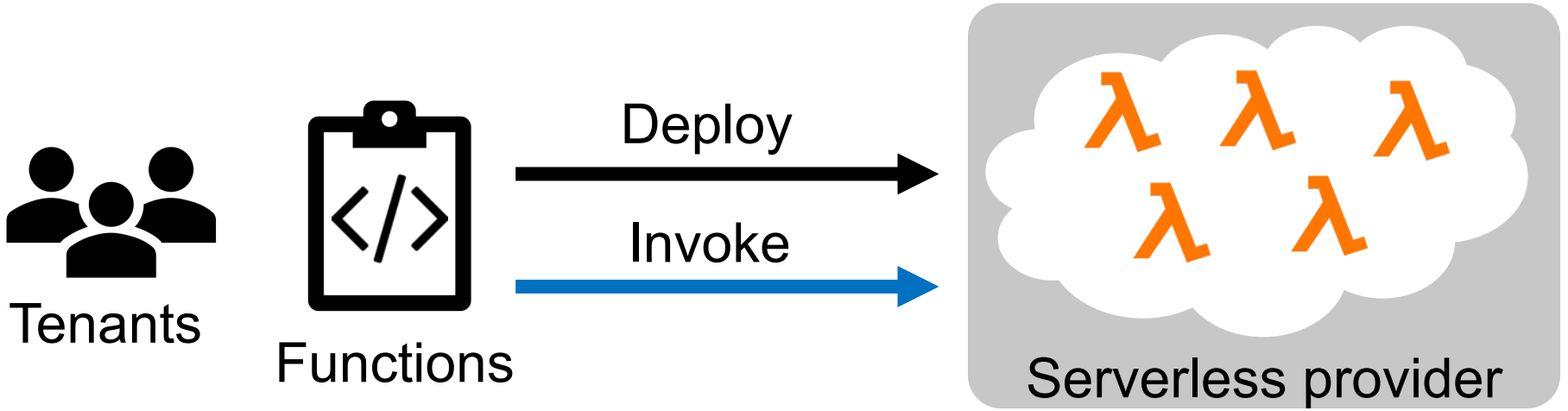
A primer on Serverless Computing

- Serverless computing enables cloud tenants to launch short-lived tasks (i.e., Lambda functions) with high elasticity and fine-grained resource billing
- Function: basic unit of deployment. Application consists of multiple serverless functions

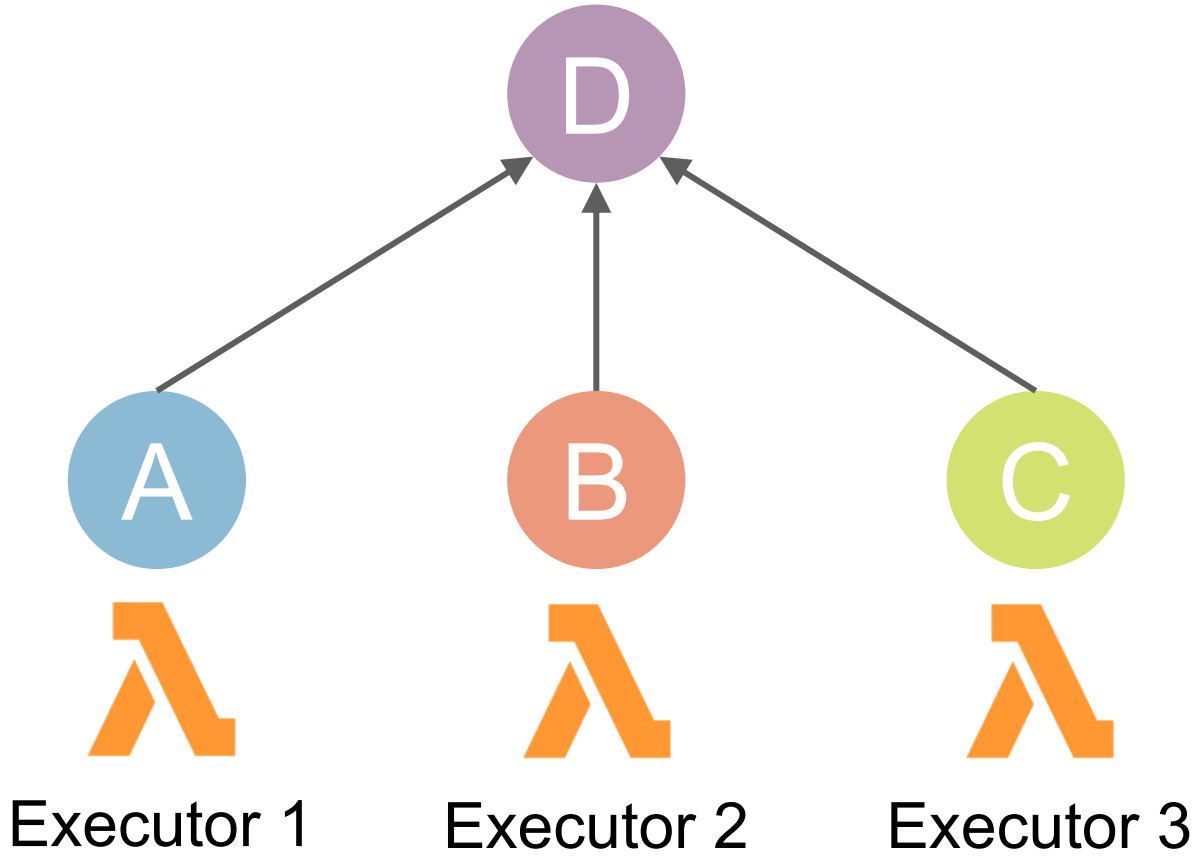


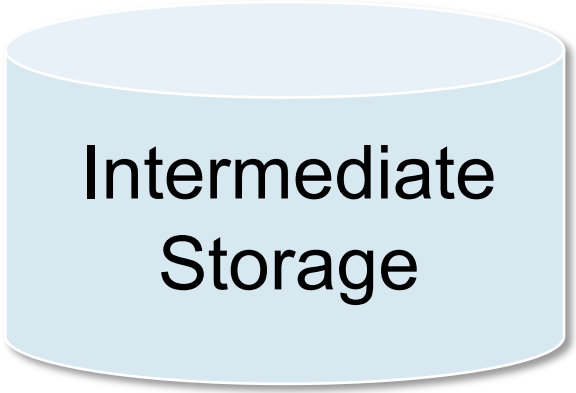
A primer on Serverless Computing

- Serverless computing enables cloud tenants to launch short-lived tasks (i.e., Lambda functions) with high elasticity and fine-grained resource billing
- Function: basic unit of deployment. Application consists of multiple serverless functions
- Popular use cases: Backend APIs, data processing…



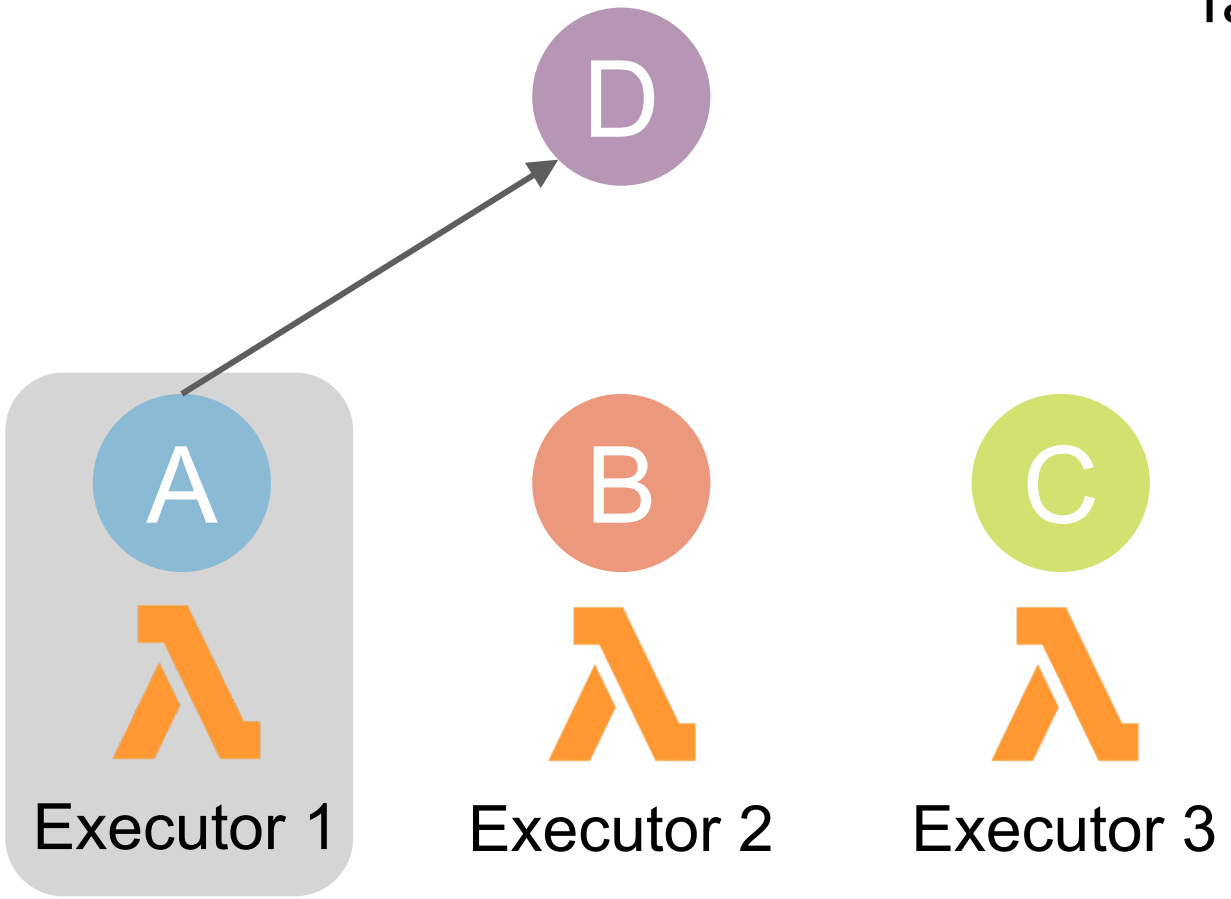
Handling fan-in



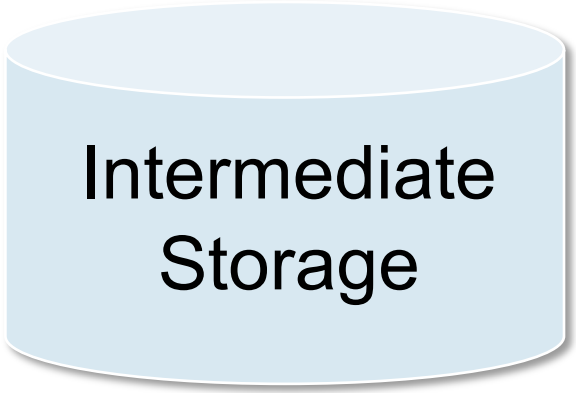


Handling fan-in

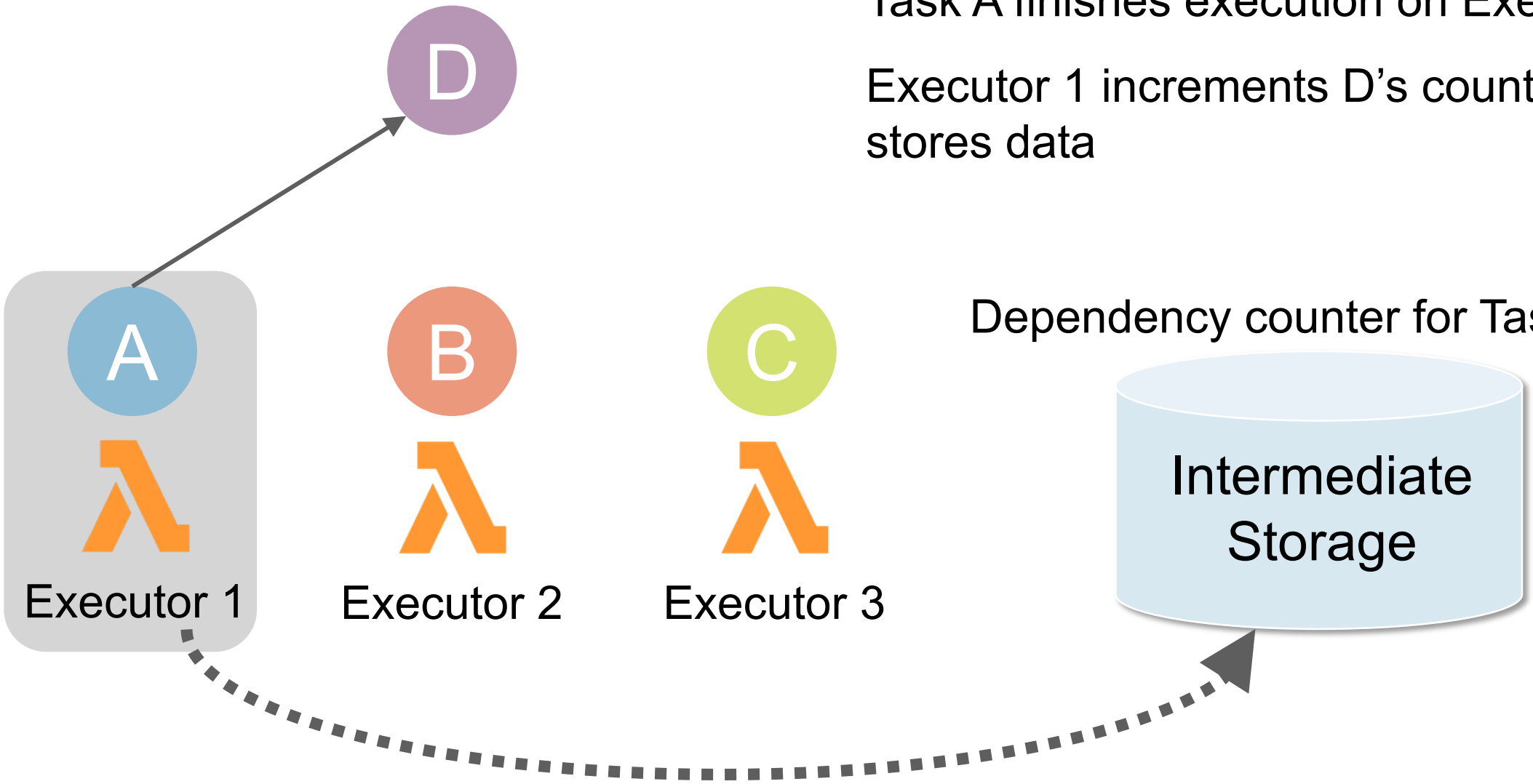
Task A finishes execution on Executor 1



Dependency counter for Task D: 0



Handling fan-in



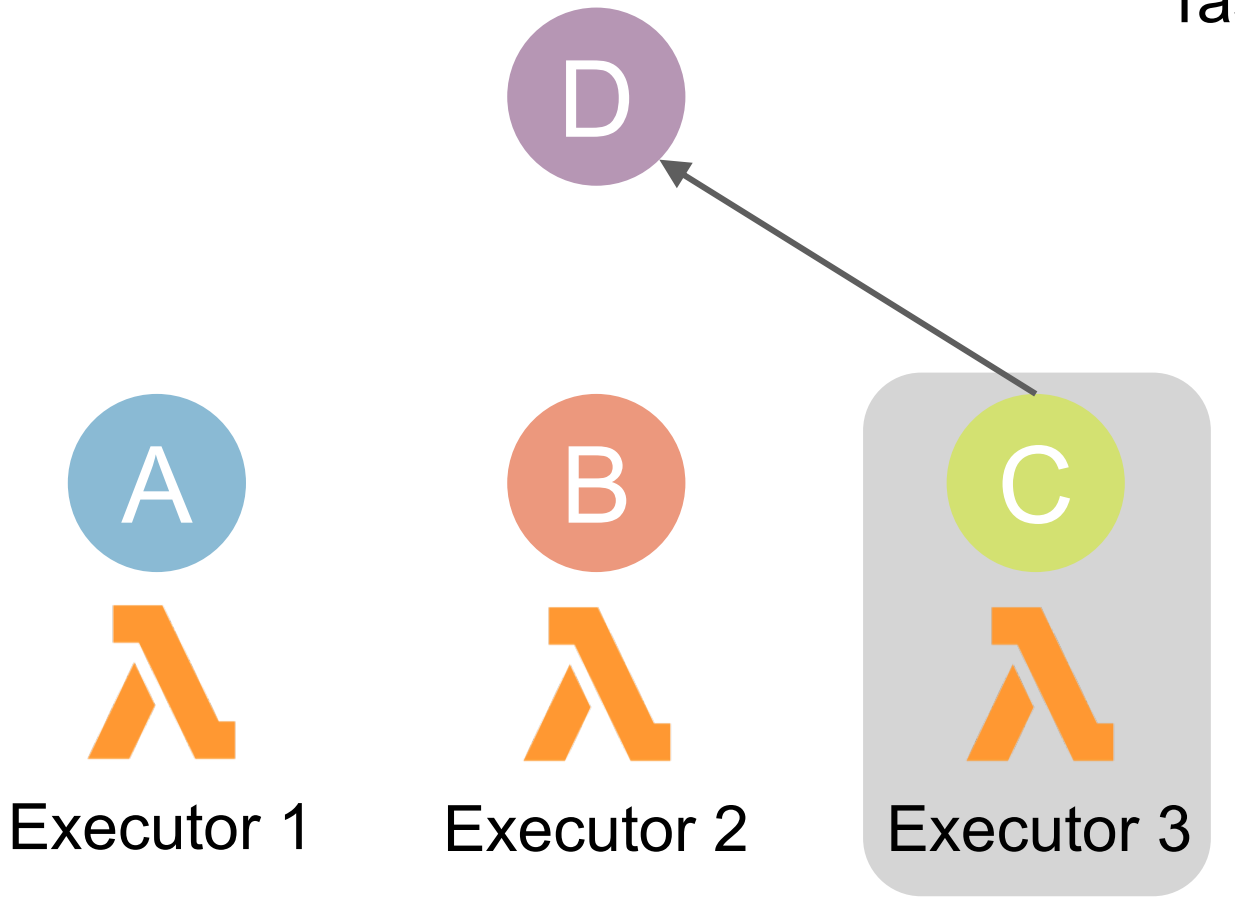
Task A finishes execution on Executor 1

Executor 1 increments D's counter and stores data

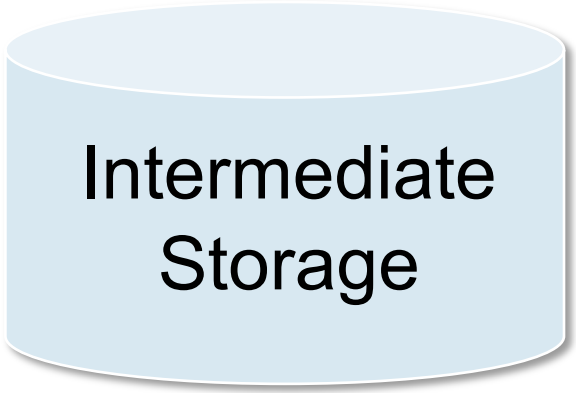
Dependency counter for Task D: **1**

Handling fan-in

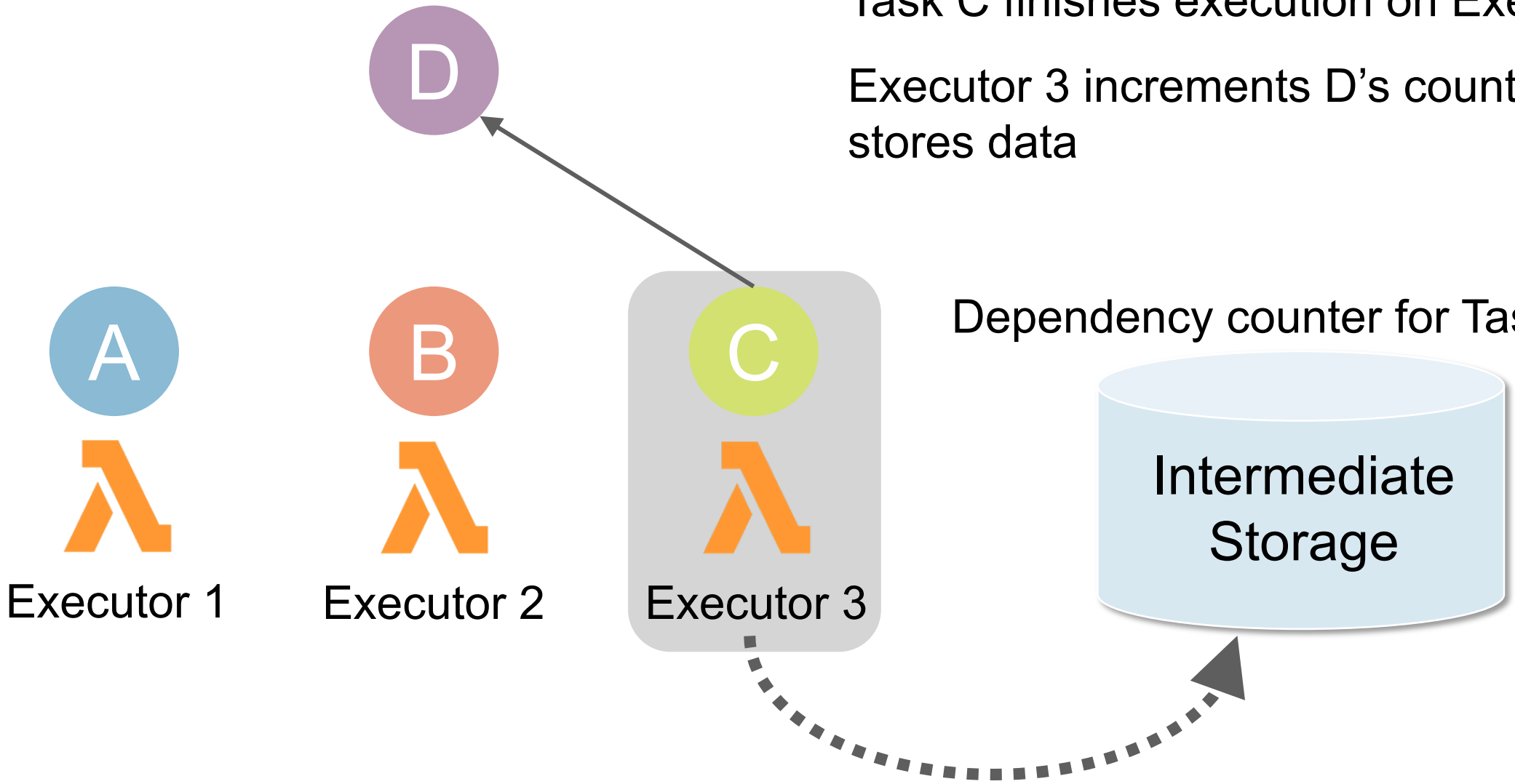
Task C finishes execution on Executor 3



Dependency counter for Task D: **1**



Handling fan-in



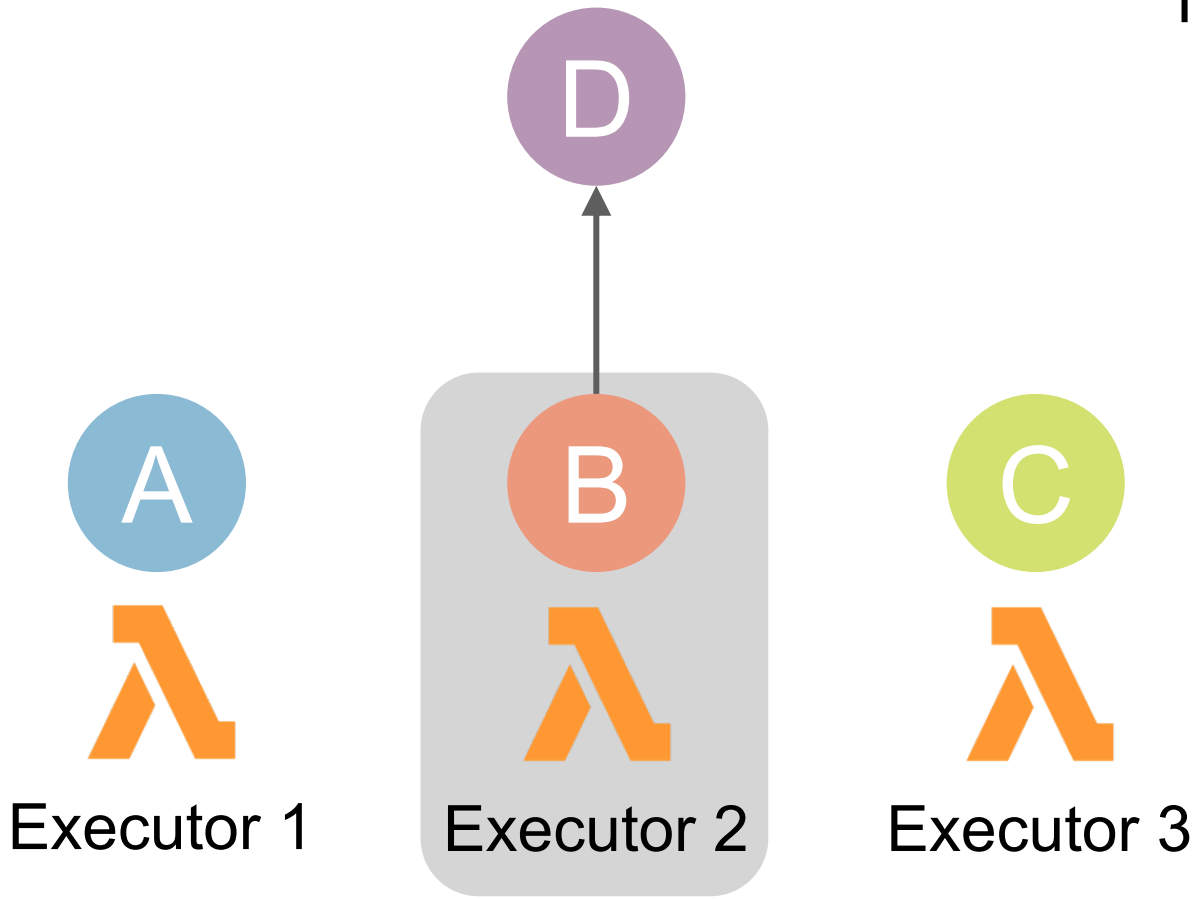
Task C finishes execution on Executor 3

Executor 3 increments D's counter and stores data

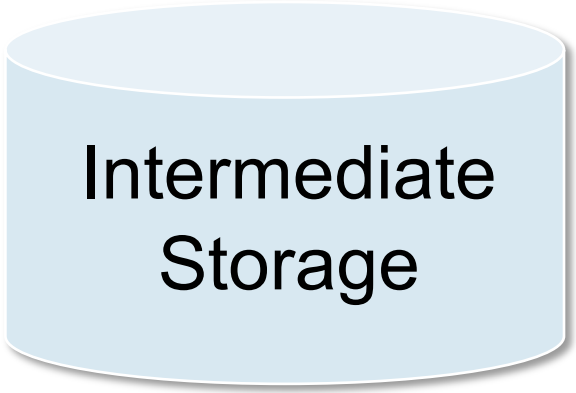
Dependency counter for Task D: **2**

Handling fan-in

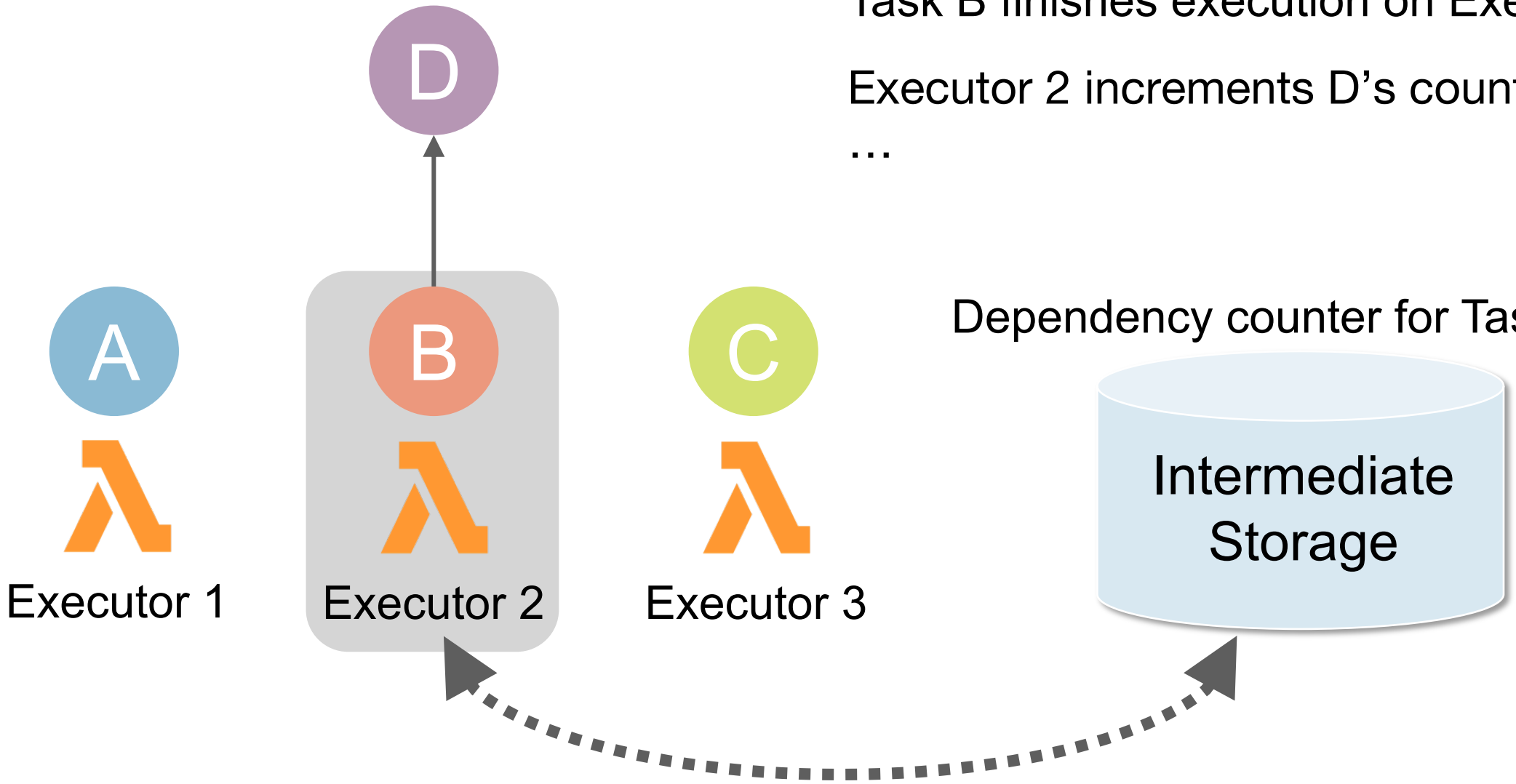
Task B finishes execution on Executor 2



Dependency counter for Task D: **2**



Handling fan-in

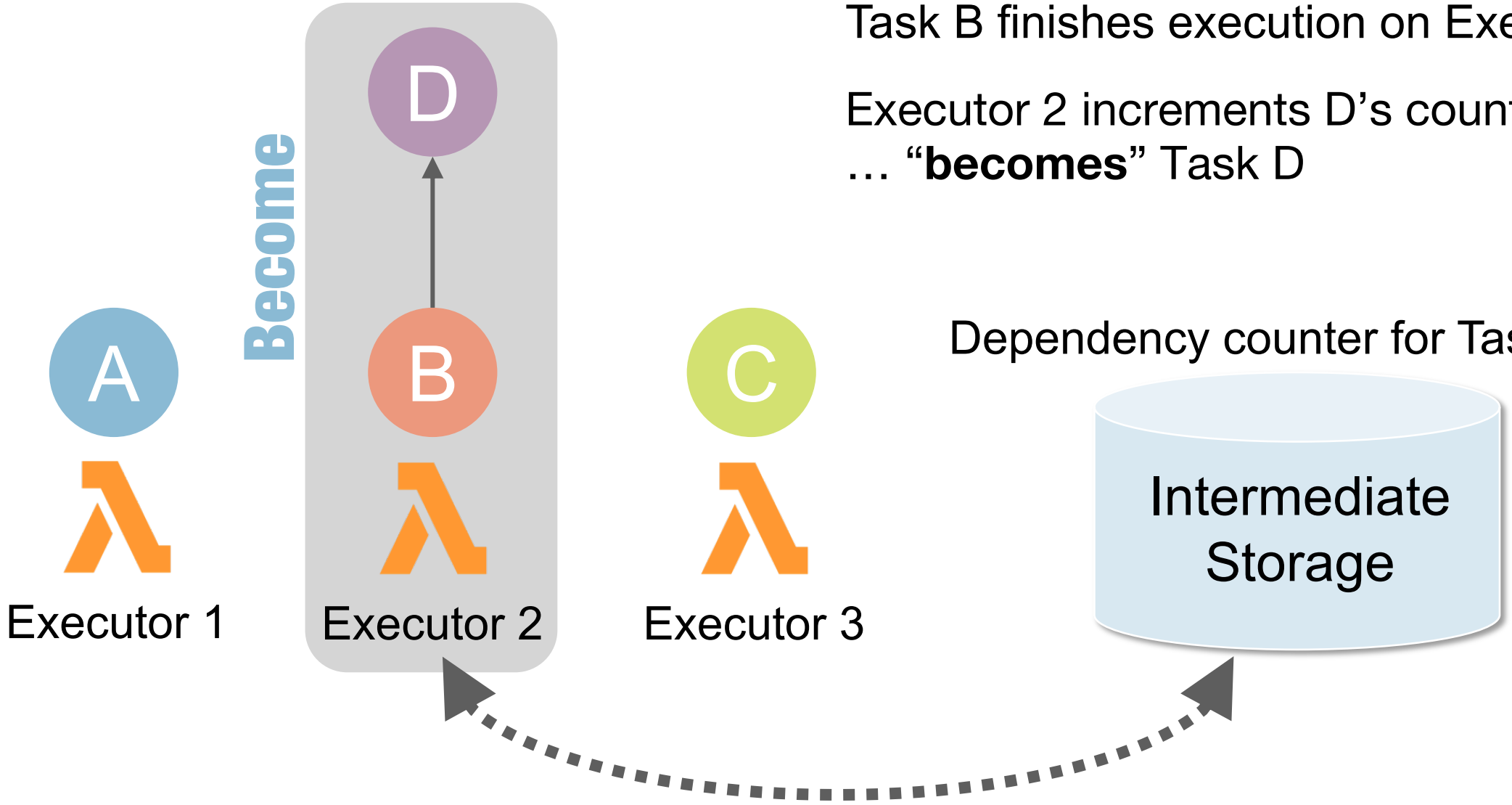


Task B finishes execution on Executor 2

Executor 2 increments D's counter and …

Dependency counter for Task D: **3**

Handling fan-in

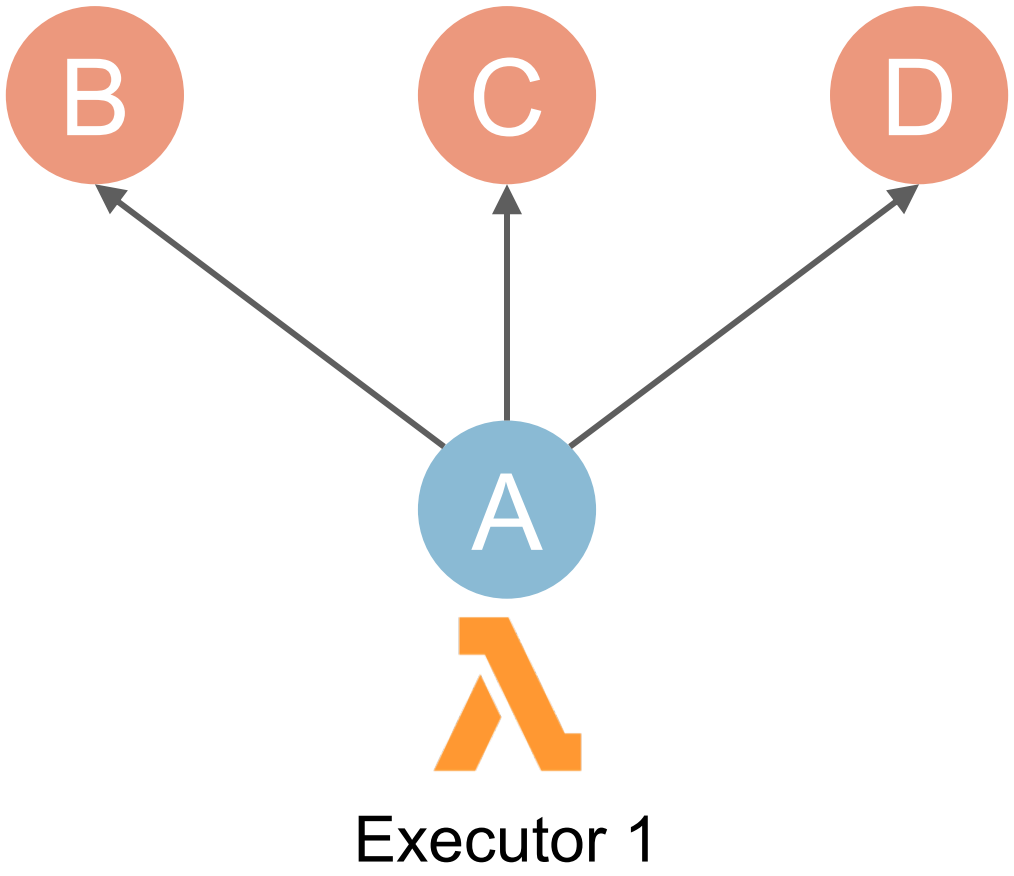


Task B finishes execution on Executor 2

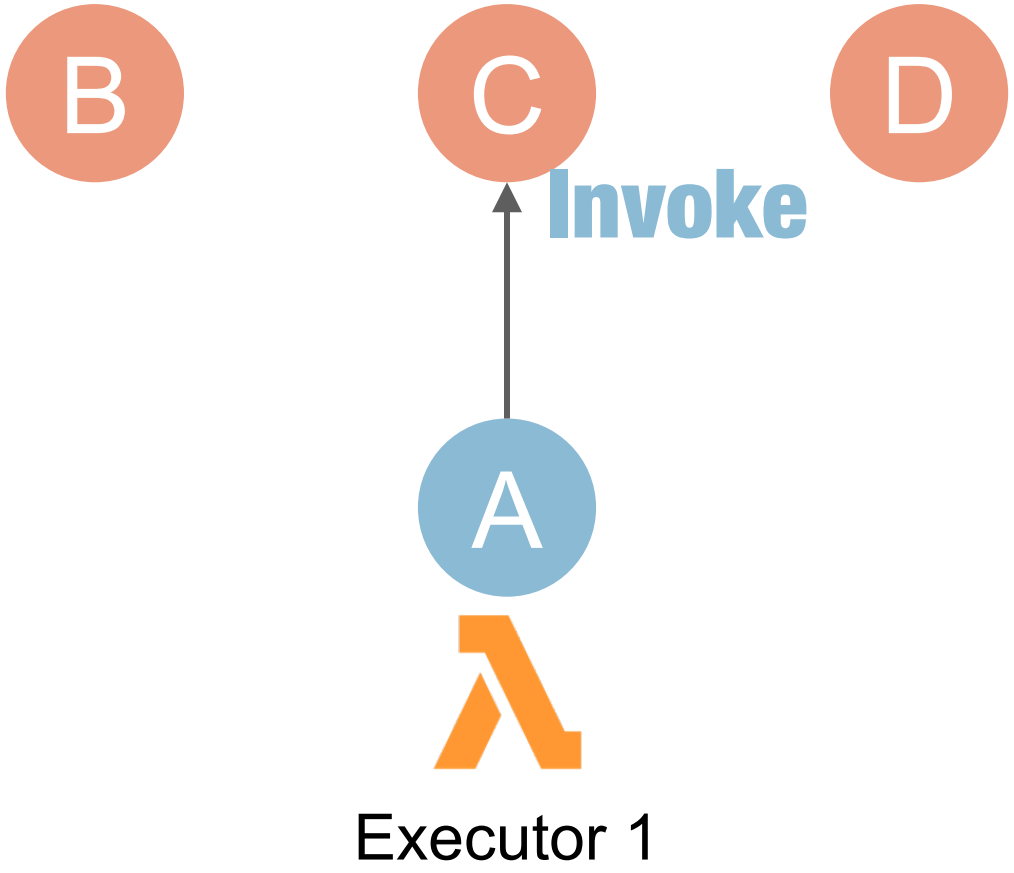
Executor 2 increments D's counter and … "**becomes**" Task D

Dependency counter for Task D: **3**

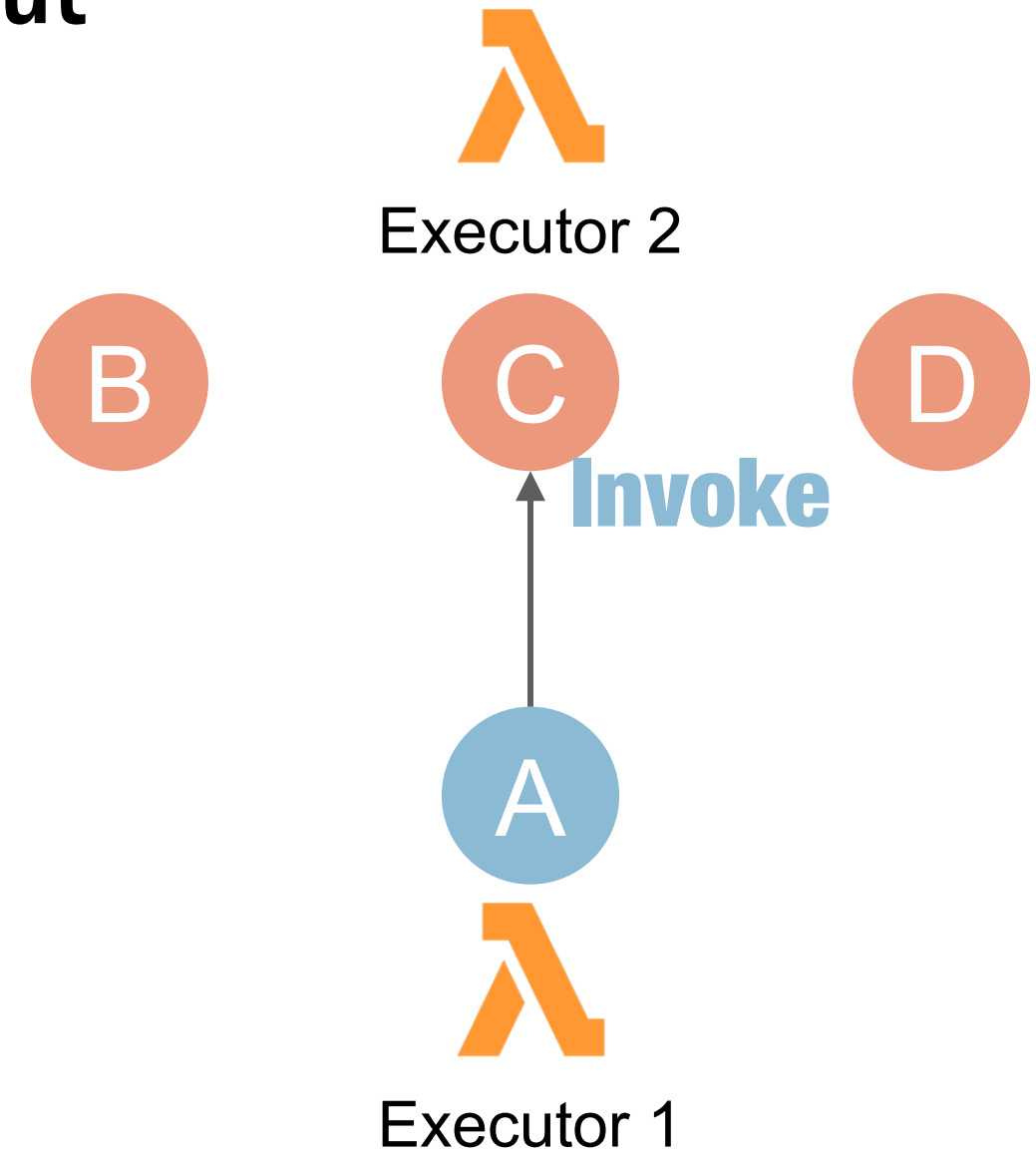
Handling fan-out



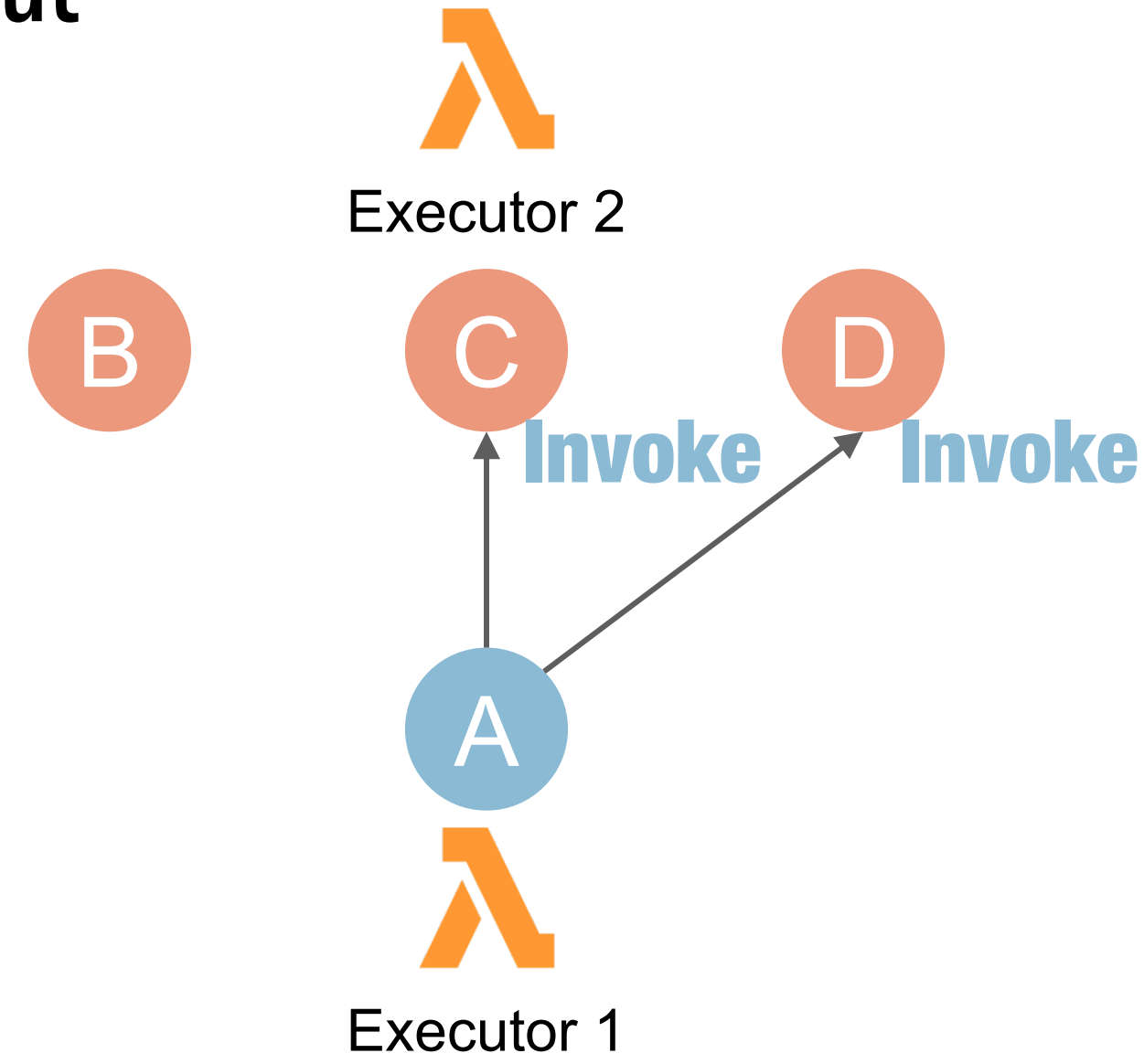
Handling fan-out



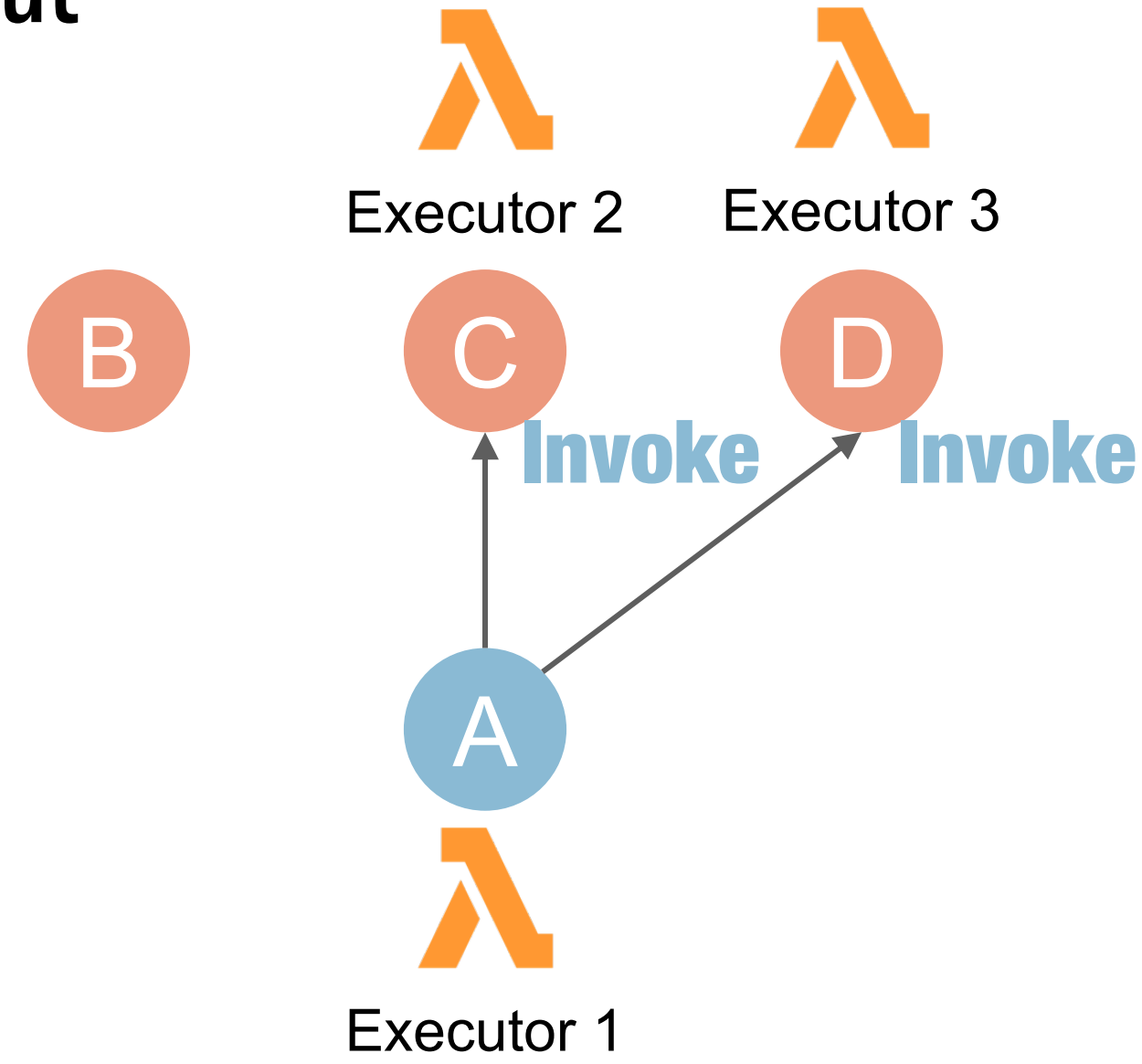
Handling fan-out



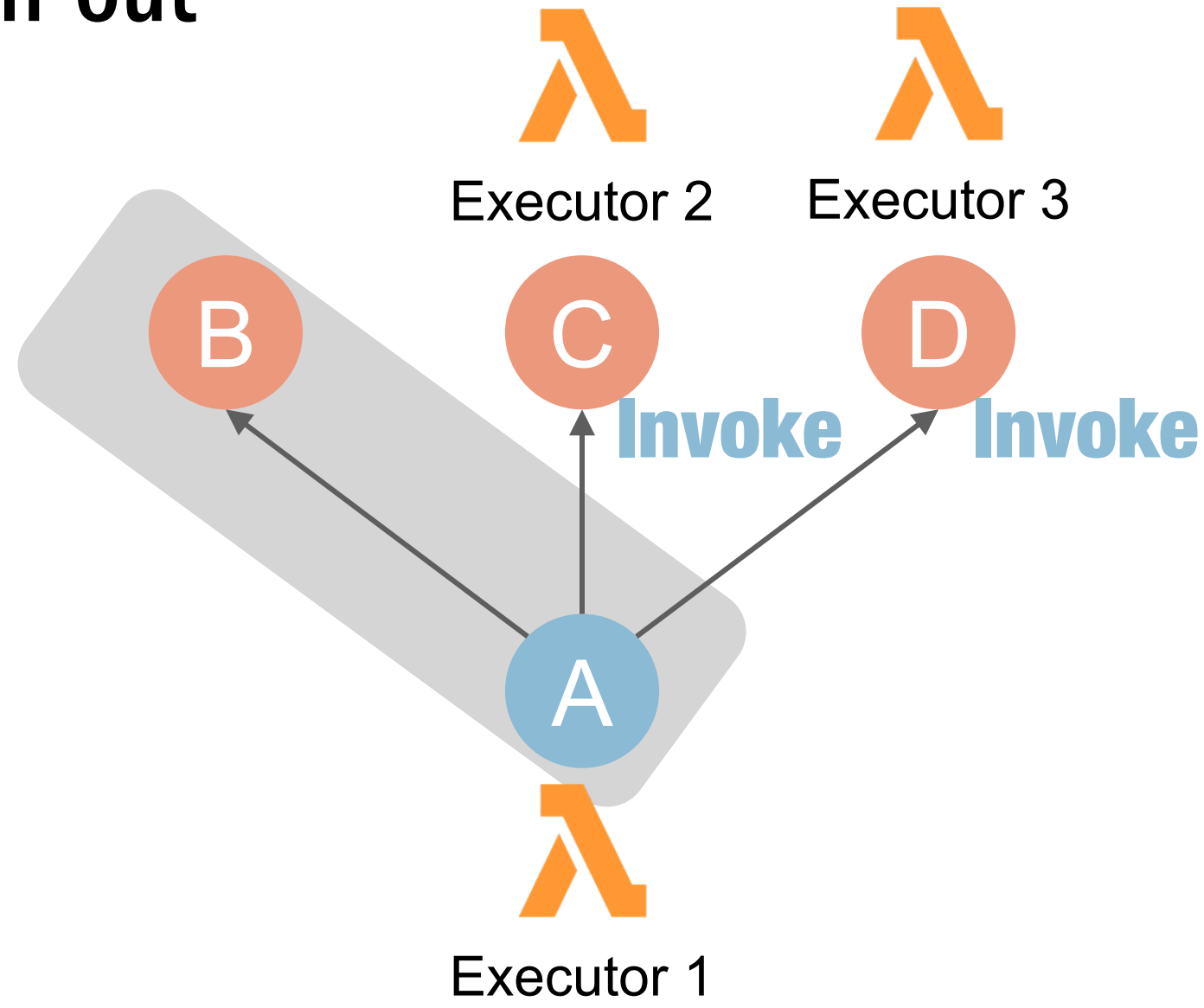
Handling fan-out



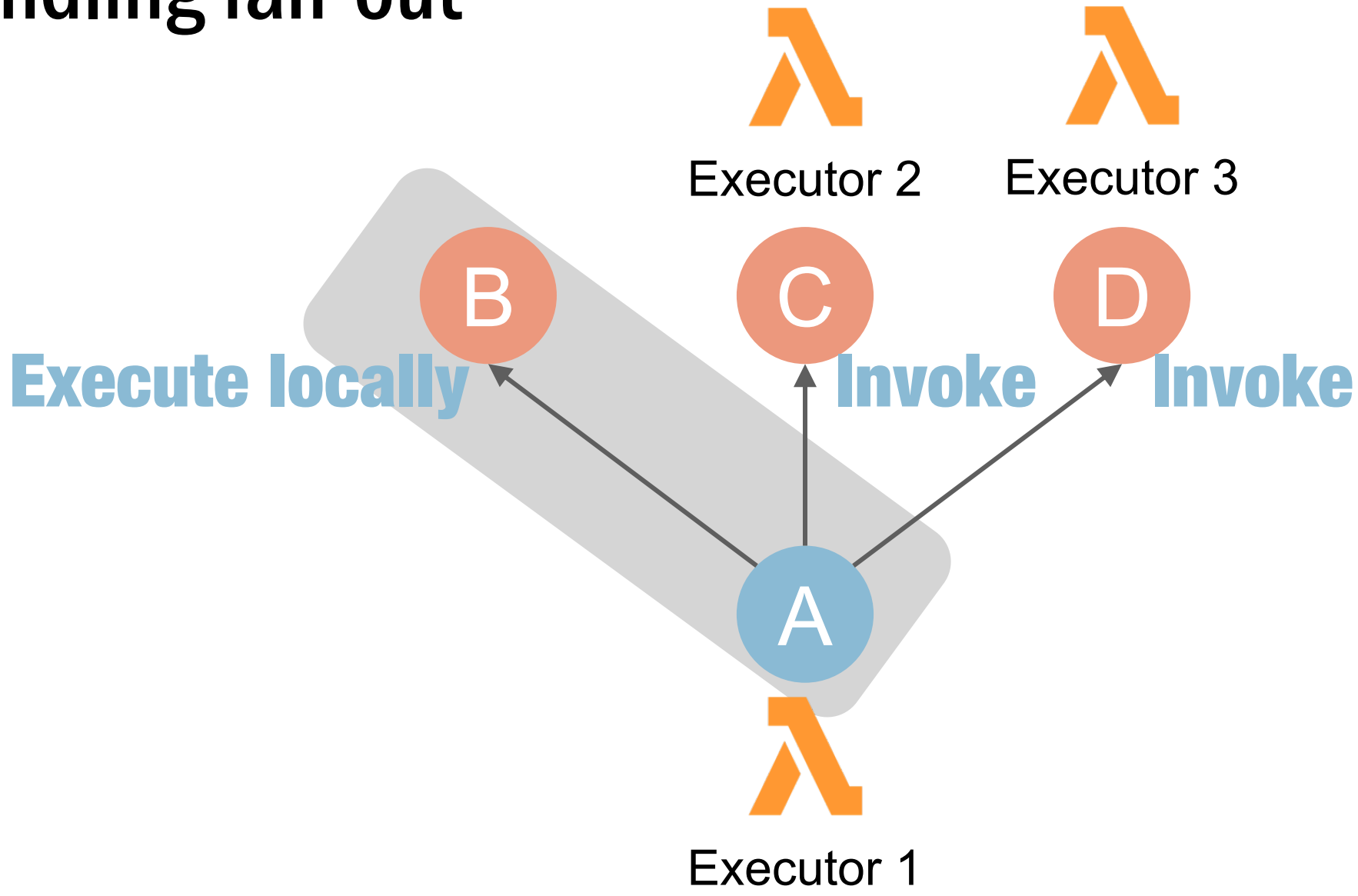
Handling fan-out



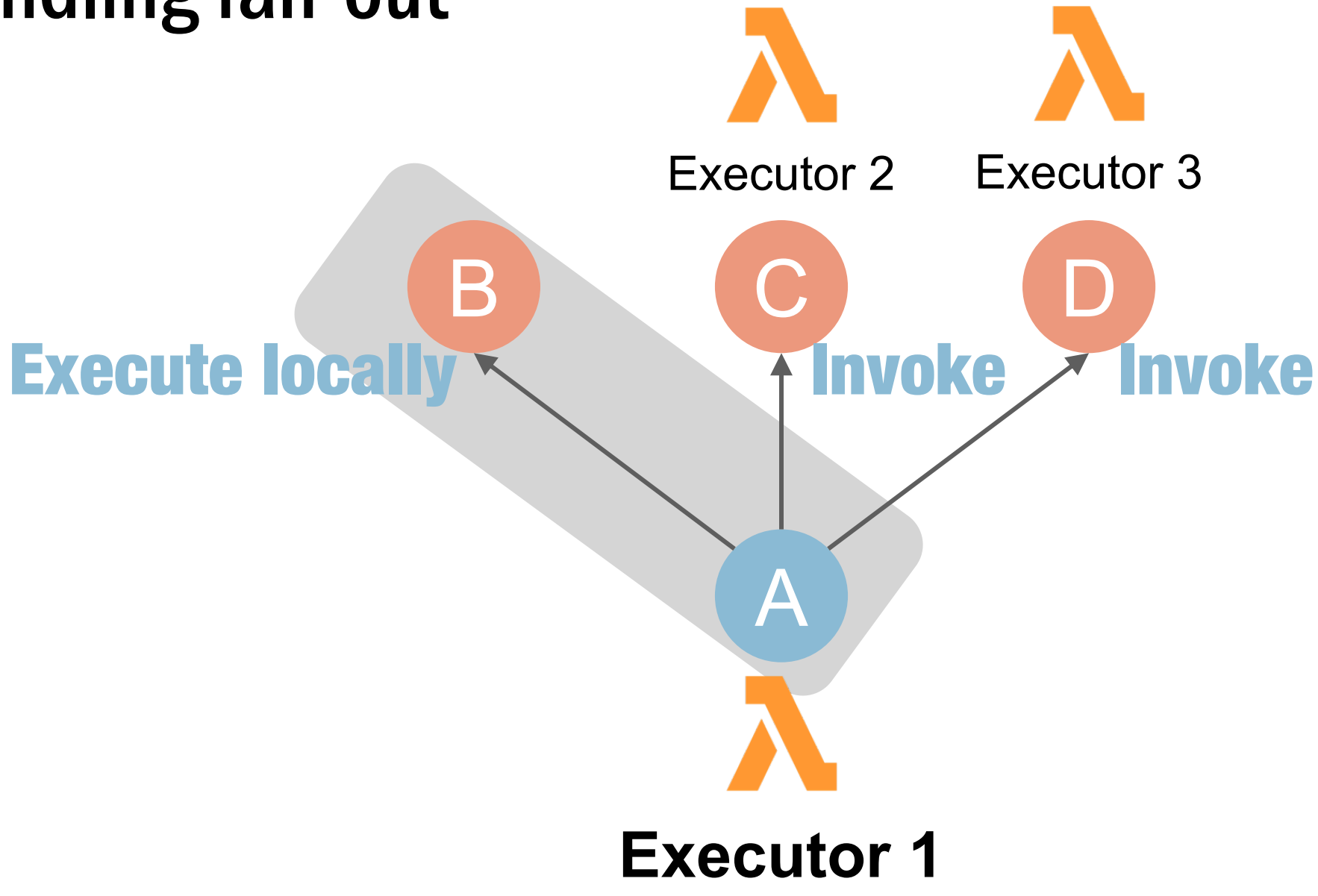
Handling fan-out



Handling fan-out

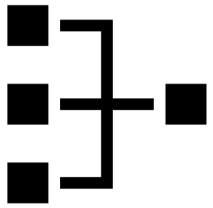


Handling fan-out



Other optimizations in Wukong

Wukong uses several techniques to enhance **data locality**



Task clustering

Eliminate intermediate data transfer by executing tasks locally

Delayed I/O

Delay performing I/O until downstream tasks are ready

Then perform task clustering on those tasks